

© 2011 Simion Venshtain

CACHE DESIGN EXPLORATION IN A GENERAL PURPOSE  
MASSIVELY PARALLEL ARCHITECTURE

BY

SIMION VENSHTAIN

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Electrical and Computer Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Associate Professor Sanjay J. Patel

# ABSTRACT

Memory model design is a major part of any modern processor architecture. There are many design choices and tradeoffs to be considered, and these often need to be tightly coupled to the processing unit's architecture. The increased popularity of massively parallel architectures has motivated researchers to further examine the memory model tradeoffs these types of architectures and their target applications present. This thesis will focus on Rigel, a 1024-core, general purpose massively parallel architecture. I will study the memory model design tradeoffs of the Rigel cluster, a subblock of the Rigel architecture, and attempt to propose a design configuration that is suitable to the unique requirements of the Rigel architecture. Rigel is an aggressive design target and requires us to focus on the area and power impact of the memory model design choices. As a result, to study the design tradeoffs, I use an approach that utilizes an RTL implementation, combined with a custom design exploration flow built on top of production quality CAD tools. This flow allows us to extract accurate power and area results for each design point and pick points that provide us with the highest performance density.

# TABLE OF CONTENTS

LIST OF ABBREVIATIONS . . . . .	v
CHAPTER 1 INTRODUCTION . . . . .	1
1.1 Rigel: A Massively Parallel General Purpose Accelerator . . . .	1
1.2 The Rigel Cluster . . . . .	2
1.3 The Rigel Core . . . . .	4
1.4 Motivation . . . . .	8
1.5 Thesis Organization . . . . .	10
CHAPTER 2 L1 INSTRUCTION CACHE . . . . .	11
2.1 Motivation for L1I . . . . .	11
2.2 Cluster Perspective . . . . .	11
2.3 Cache Design . . . . .	12
CHAPTER 3 L1 DATA CACHE . . . . .	20
3.1 L1D Write Policy . . . . .	20
3.2 L1D Design . . . . .	22
3.3 Cluster Interconnect Impact . . . . .	34
CHAPTER 4 CLUSTER INTERCONNECT AND CONTROL . . . .	35
4.1 Cluster Interconnect . . . . .	35
4.2 Interconnect Control . . . . .	39
CHAPTER 5 CLUSTER CACHE . . . . .	42
5.1 Splitting Instruction and Data . . . . .	42
5.2 Cache Design - Reuse of the L1D Design . . . . .	43
CHAPTER 6 EXPERIMENTAL SETUP . . . . .	46
6.1 Synopsis Toolflow . . . . .	46
6.2 Simulation Flow . . . . .	47
6.3 Design Exploration Flow . . . . .	48
6.4 RTL Style . . . . .	50
6.5 SRAM Hard Macros . . . . .	52

CHAPTER 7	EVALUATION . . . . .	53
7.1	Design Power and Area Evaluation . . . . .	53
7.2	Cache Verification . . . . .	54
7.3	Design Space Exploration . . . . .	54
7.4	Evaluation Metrics . . . . .	55
7.5	Benchmarks . . . . .	55
7.6	Results . . . . .	60
CHAPTER 8	CONCLUSION . . . . .	66
REFERENCES	. . . . .	68

# LIST OF ABBREVIATIONS

CMP	Chip Multi-Processor
CUDA	Compute Unified Device Architecture
DUT	Design Under Test
FIFO	First In First Out
FSM	Finite State Machine
GPU	Graphics Processing Unit
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
MIMD	Multiple Instruction Multiple Data
MLP	Memory Level Parallelism
MSHR	Miss Status Handling Register
RTL	Register Transfer Language
SAIF	Switching Activity Information Format
SPMVM	Sparse Matrix Vector Multiply
SRAM	Static Random Access Memory

# CHAPTER 1

## INTRODUCTION

### 1.1 Rigel: A Massively Parallel General Purpose Accelerator

Massively parallel computing has gained popularity in recent years in both the commercial and scientific domains. Multiple machine architectures have been proposed [1],[2],[3],[4] to extract high parallelism from an application. Each architecture has advantages, disadvantages, and tradeoffs when considering power consumption, total cost, unit cost, and performance. One subset of such architectures are hardware accelerators, entities designed to perform a specific class of functions faster than is possible on a general purpose CPU. Accelerators are designed for computationally intensive software and take advantage of characteristics of the target application to provide benefits including higher performance, lower unit cost, and lower power compared to general purpose CPUs. Accelerators can have varying architectures based on the computation needs and can be customized to utilize stream-based datapaths, custom function units, vector processing, and specialized memory systems. One of the most common hardware accelerators found in computers today is the graphics processing unit (GPU). NVIDIA GPUs have gained a lot of interest from the scientific community [1], providing a relatively easy and inexpensive introduction to massively parallel computing. NVIDIA's CUDA [5] (Compute Unified Device Architecture) programmable platform can be applied to a variety of domains with high data parallelism that requires high performance. CUDA, however, is not without limitations. Software managed coherence and poor performance on highly divergent control flow make this platform ill suited for some applications. A different approach to massive throughput on a single chip is to use many simple in-order MIMD (multiple instruction, multiple data) cores on a single chip, with

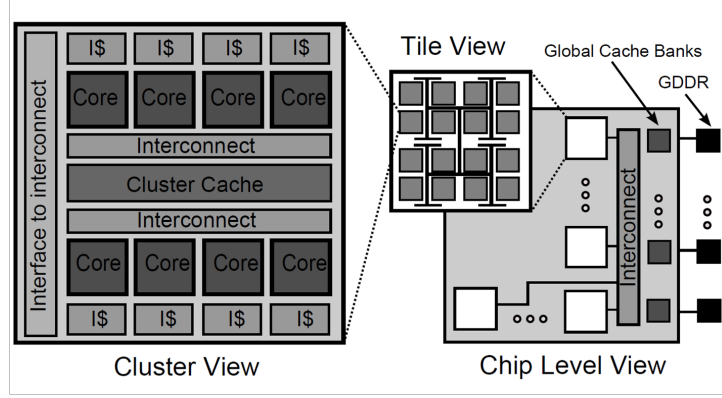


Figure 1.1: Rigel architecture

a mix of hardware and software coherence to alleviate the GPUs limitations. The proposed architecture, Rigel [6], is 1024 core CMP. Rigel is throughput oriented architecture, consisting of many simple processing elements. The idea is that a single unit of work performance will be poor compared to a modern CPU; however, the abundance of work and cores will produces better performance overall. Figure 1.1 illustrates the Rigel architecture. Rigel is composed of 16 tiles, with each tile composed of 16 clusters. The cluster is an entity composed of eight cores and cluster level cache connected over a local interconnect. The clusters are attached over a global interconnect to a global cache. The global cache serves as the synchronization point for the entire chip. However, for scalability reasons, there is no hardware coherence between the cluster cache and the global cache, thus the reliance on software coherence.

## 1.2 The Rigel Cluster

The Rigel Cluster consists of three main components: eight cores, a cluster level cache and an interconnect to connect the two. Some design elements in the Rigel cluster are fixed such as the core architecture and the 32 kB size of the cluster cache, which I also refer to as the L2 cache. However, this work will explore the design tradeoffs for the unexplored parts of the cluster design, which are the interconnect and the cache design. The design points I explore can seem very discrete and extreme on the spectrum of the design space and one might argue that the optimal design lies somewhere within that



spectrum. I recognize that criticism and rationalize my work by arguing that looking in the entire design space is well beyond the time limitation of this work. As will be discussed later, my goal is not to be exhaustive in my design exploration, but to explore a cross section of the design space and to present an effective foundation for future exploration with an effective methodology.

### 1.2.1 Cluster interconnect

There is a wide variety of interconnects, each having complexity and bandwidth tradeoffs. We can choose to use anything from a simple interconnect such as a bus or a crossbar [7], or a more complex design such as a ring or a packet based network [8]. Complex interconnects are more appropriate for larger networks such as a chip-wide interconnect. For the interconnect within a single cluster a simple interconnect is more appropriate. Hence, I have chosen to consider two simple yet very different interconnects, bus and crossbar. Bus connects all the cores to the cluster cache over a single point connection. A bus interface provides very low complexity in terms of area, power and design; however, contention can lower the effective bandwidth and hurt performance. On the other side we can choose a crossbar interconnect, a significantly more complex interconnect. The crossbar connects  $N$  cores to  $M$  banks in the cluster cache, which makes the wire complexity  $N \times M$ , and provides  $M$  connections to the cluster cache. The crossbar interface has maximum bandwidth similar to that of the bus, but is much less susceptible to contention.

### 1.2.2 Cache hierarchy

When we make a decision regarding the cache hierarchy, we have to tightly couple it with our decision on the interconnect, since the cache hierarchy will affect the cluster interconnect usage. I couple each discrete interconnect with a discrete cache hierarchy. In case of a bus I will look into a two level hierarchy, where each core has a privatized L1 and all the cores share a centralized L2 (cluster cache). The second hierarchy is a banked shared cluster connected to the cores with a crossbar.

### 1.2.3 Cluster cache design

In addition, it is important to explore the many design options and optimizations for the cluster cache. It is difficult and beyond the scope of this work to explore all of them; hence, I chose to look into a single design tradeoff that I think will have significant impact on performance. I will evaluate whether the cluster cache should be a blocking or nonblocking design. A nonblocking design is more complex to implement and more hardware demanding than the traditional blocking design; however, it will not block on a cache miss, and in effect supports resolving multiple misses simultaneously. As mentioned before I will keep the cluster cache size constant at 32 kB. Also, a crossbar interconnect will require banking the cluster cache, which requires duplication of the control logic and some hardware structures regardless on the actual design.

### 1.2.4 Other design factors

Other design choices that relate to the memory system architecture are the memory consistency model and the cache coherence. Cache coherence is the issue of making sure that all the execution elements in a system have the same view of the data in the system. Reference [9] presents a detailed survey of cache coherence protocols. The more cache levels and execution units the system has, the more challenging it is to maintain the system coherent. The memory consistency model deals with the guarantees you put on your system regarding the ordering of memory operations with respect to the processor itself and with respect to the whole system. The different consistency models and the tradeoffs between them are discussed in [10]. Due to time limitation, I made fixed design choices about the above points. I chose to have a processor consistency model with software coherence.

## 1.3 The Rigel Core

As stated previously, the Rigel architecture is throughput oriented. The goal is to have many processing elements and an abundance of parallelism to provide work for all of them. In order to achieve the aggressive Rigel target

design of 1024 cores, the cores need to have low area and power footprints. Low area/power requirements result in a simple core design.

The Rigel core block diagram can be seen in Figure 1.2. The core RTL passed through synthesis and place-and-route flow. The Rigel core layout can be seen in Figure 1.3. The core is 2-wide in-order execution and retire. The core is in essence a slight variation of the classical 5-stage pipeline.

The pipeline stages are as follows:

**Fetch** The fetch stage fetches instruction from the L1 instruction. The core is 2-wide so the fetch stage fetches 2 instructions every cycle. The fetch stage will stall until the decode issues both previous instructions.

**Decode** The decode stage will decode instructions, read operands from the latch based register file or from the bypass network, check for dependencies with the second instruction, and check the scoreboard for dependencies with later stages. The decode will issue a maximum of two instructions to the execution stage. The decode stage will stall on data dependencies or in case of a branch, since we do not employ branch prediction.

**Execution** There are three execution units: Integer Unit, Memory Unit and Floating Point Unit (FPU). The FPU unit is four cycle latency. For reasons explained later in this section the other two execution units will have bubble stages to match the FPU latency. The memory stage will communicate with the L1 data cache. This interface and operation will be discussed in detail in the next chapters.

**Write-Back** The write-back stage will write back to the register file. There are two write ports on the latch based register file, which will support two write-backs in the same cycle.

### 1.3.1 In-order execution

The Rigel core is an in-order execute and commit. Extensive stall logic connecting inter- and intra-execution units enforces in order commit. The bubble stages in the memory and integer execution unit simplify the stall logic. For each stage the stall logic needs to worry about instructions in the

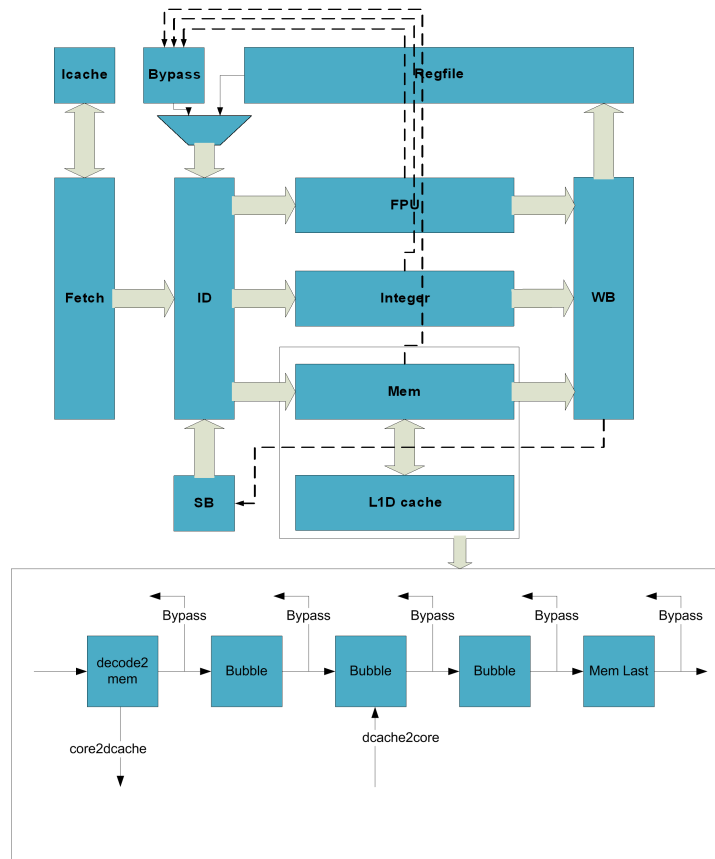


Figure 1.2: Rigel core architecture

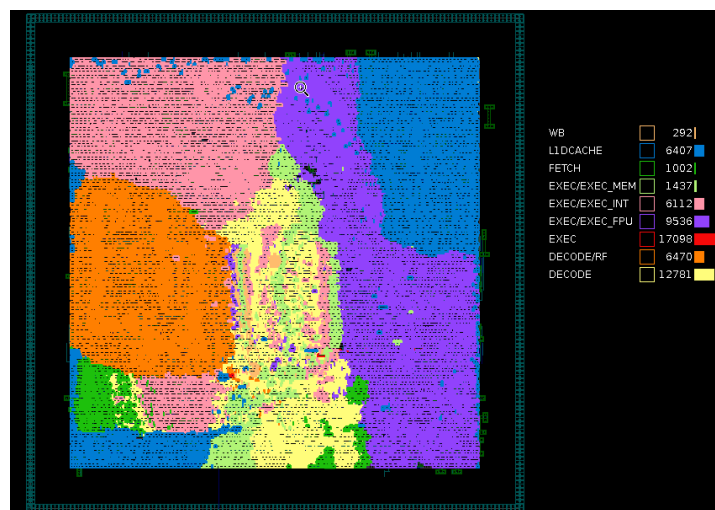


Figure 1.3: Placed and routed core

same and following execution cycles. This eliminates a long critical cycle in stall signals, where stall signals had to propagate from the last to the first stage. By equalizing the execution, it is simple to track the order of the instruction. The instruction that is ahead in the pipeline latches is the older. As a result of using this method to enforce in-order execution, a memory stall in the first memory stage will stall the entire execution in core. A more elegant solution is to stall the instruction only at the last stage, which requires a more complex cache implementation.

### 1.3.2 ILP optimizations

Although very limited by the simplicity of in-order execution and retire, the core still makes an attempt to expose ILP using several optimizations. The main optimization in the core is the full bypass network, which forwards the results from all the stages of each execution unit. The bypass network prevents the bubble stages from adversely impacting the performance. Also, the bubble stages themselves serve as an ILP optimization, by allowing buffering of execution in case of a memory stall in the cycle of the memory unit. In addition, I employ an optimization that allows us to use the bypass network as a renaming method, removing output dependencies. This is achieved by adding a ready signal for each bypass packet.

I will evaluate the impact of these core optimizations on the performance, power and area of the entire system. To fairly evaluate this design feature I will do my best to remove the data dependencies in the core using loop unrolling and compiler optimizations.

### 1.3.3 L1 data cache design

The part of the core that has the biggest effect on the overall performance is the L1 data cache. Having a nonblocking cache that supports multiple requests in flight and allows the core to stall only at the last stage of the memory pipeline can allow the core to execute additional independent instructions. Once we encounter a memory stall the entire core will stall, not allowing any new instruction to execute. With a naive blocking cache, the memory request will stall in the first stage of the memory unit, stalling all

core execution. Nonblocking cache elevates the stalling by allowing memory instruction to flow until the last stage of the memory pipe. Using the current nonblocking implementation, the most MLP can be extracted by scheduling enough back-to-back independent memory requests to result in a miss. Any other mix of instructions will not achieve the maximum MLP, but will potentially still perform better than a naive blocking cache.

#### 1.3.4 Ability to support MLP in a simple core

Modern microprocessors employ advanced techniques to expose ILP that eventually results in MLP. Using reorder buffers and renaming techniques, they support out-of-order execution, which allows the processor to execute independent instruction on a cache miss. These advanced techniques allow us to hide the memory latency by executing other independent instructions and, as a result, exposing future memory requests as early as possible; then we can start resolving those requests in parallel with previous ones. Although the Rigel core is area/power efficient, its simplicity casts doubt on the ability to expose any ILP and MLP. Carefully written software and proper compiler optimization are required to provide the core any chance to achieve MLP.

### 1.4 Motivation

With this work I will attempt to explore tradeoffs in several design points for the cluster memory system. A careful measurement and investigation is required when it comes to an aggressive design target such as Rigel. Such a design is power/area sensitive and each design decision in the cluster level will affect all the clusters in the design. In this work I stress the point that the components that affect the memory model are not disjoint, and careful co-design is required. References [7] and [11] make the same claim, but have taken an analytic approach to explore the design space. As a result, studying a single design component at a time—such as the core, cache or interconnect—is flawed and is a clear path to finding a local optimum design rather than a global optimal design point. In order to accurately study the tradeoffs of a single component, its features need to be supported by the entire stack. I have two main goals with this work. First, I intend to study several

design tradeoffs in the cluster architecture and their effect in the global perspective. The second goal is to develop a reusable, RTL based methodology that can effectively and accurately produce performance/power/area results.

#### 1.4.1 Finding design tradeoffs

Studying each design tradeoff for every component needs to be done from a system-wide point of view. For example, to evaluate the cache architecture we have to consider the entire memory model. To fully evaluate the crossbar architecture we need to be able to hide the higher latency, and in order to hide the latency we need an effective nonblocking cache design. Lastly, in order to exploit this nonblocking cache we need to expose enough MLP. The later can be done either dynamically or by reorganizing the code.

Previous work studied cache design tradeoffs in power, area and performance [12],[13],[14],[15], but they did not measure the core and interconnect design impact on the cache performance. Other work considered co-design of cache/interconnect [7],[11] to find the system-wide tradeoffs; however, they did not consider a throughput architecture with simple cores and did not perform design exploration with core optimizations in mind. Past work [16],[17] on a Rigel-like architecture studies the tradeoffs of the entire stack of the system; however, they did not study the effects of cache/interconnect impact as comprehensively as this work, using RTL implementation. The metrics to evaluate the design tradeoffs in Rigel—a high throughput, power/area sensitive design—are not the traditional single dimensional metrics. In such a design we care about maximizing performance density with respect to both power and area; hence, the best design will be the one that maximizes performance/power and performance/area.

#### 1.4.2 Design exploration methodology

This work goes beyond the goal of finding a good design for the cluster level interconnect and caches; in fact, I cannot claim that I will find an optimal design point. My design space exploration is not meant to be complete or exhaustive, but simply an exploration of some major CMP design space considerations throughout the stack of the design. A major part of my work is

to develop a methodology for design space exploration using an RTL toolflow, allowing detailed study of performance/power/area tradeoffs and laying the foundation for further design space exploration. In a way, the cluster design exploration is a way to demonstrate the flow and show its effectiveness.

Some previous work on design space exploration for performance, area and power tradeoffs exists. Some work uses analytical models [18]; others use Tensilica's synthesizable embedded processor framework [19] to find hardware-software co-design [20], but remain limited to a fixed ISA and limited RTL flexibility. Previous research on the Rigel architecture [16],[17] used a mixed analytical and RTL approach to speed up the design exploration time, which was appropriate to converge on major architectural details, but will be less accurate for detailed architectural design decisions.

## 1.5 Thesis Organization

The first part of my work will discuss the L1 cache design and implementation. First I will discuss the L1 instruction cache part of my work. Following that I will proceed to discuss the L1 data cache. The L1D section will talk about the write policy tradeoffs and policy decisions I have made for the Rigel cluster. I will continue to discuss the L1D design, which includes two designs: a simple blocking cache design and a complex nonblocking design. I will discuss the tradeoffs of the two designs. I will continue to discuss the interconnect interfaces, control logic and arbitration policy for each cluster design. I will finish my implementation discussion by presenting the cluster cache design, which will be a naive implementation based on the L1 data cache. After describing the design I will discuss my experimental environment, the design exploration methodology and RTL style I employed to study the cluster design. Following by that I will present my experiments and results for several data oriented benchmarks.



# CHAPTER 2

## L1 INSTRUCTION CACHE

As part of my work I wanted to complete the core to the extent that it had all the major area and power components. As a result, I also worked on implementing and verifying the L1 instruction cache. The L1I has unique design tradeoffs.

### 2.1 Motivation for L1I

When investigating the design space of a core in Rigel, we have made a design choice to have a privatized instruction cache. The reasoning behind that design choice is based on several realizations. First, it is critical for any design to have a short latency instruction fetch. Modern complex processors have a multicycle pipelined fetch unit to reduce cycle time. This fetch unit is usually complemented with advanced branch prediction, which minimizes the branch misprediction cost by reducing the number of mispredictions. Our simple core makes no attempt to employ branch prediction, which makes branches always mispredict. As a result, we need to start fetching from the right address as fast as possible and that requires having a single cycle hit L1I.

### 2.2 Cluster Perspective

I will keep the instruction cache hierarchy separate from the design space exploration. To achieve that I will make a separate L2 for instructions with a bus based interface. That will allow me to focus on the data cache and isolate the impact on performance, area and power of data cache design tradeoffs.

## 2.3 Cache Design

The core and cache are connected by two separate unidirectional buses. The request bus consists of a 32 bit address bus and a single bit valid bit. There is no need for a message bus since the only request the core will make is a fetch. The L1I response bus consists of a 32 bit address, 64 bit data and a single valid bit. This unidirectional interconnect is simpler to implement and allows us to issue a request and receive a response at the same cycle.

The nature of the application the Rigel architecture targets does not require a highly complex or large L1 instruction cache. As a result I have decided to implement a small direct mapped cache. In the result section I will explore the effect of varying the L1I cache size and show the lack of impact of the L1I on the overall performance of the system. In other words, we quickly saturate the benefit of increasing the size of the L1I.

We choose not to consider self-modifying code application in Rigel, which makes all instruction data read-only. As a result the instruction cache is much simpler than the data cache in terms of cache coherence and consistency. In other words we get cache coherence for free and we do not care about cache consistency since reordering reads does not adversely affect cache consistency. If the cache is available and the core has a new request, the core will put a valid request on the request bus. In the next cycle the request will be latched into the cache and SRAM latch and be checked for a cache hit. If a cache hit occurred, the cache will respond to the core with valid data. If the request is missed in the cache, the cache will assert the request bus signal, indicating it needs to issue a request to the next level cache. The request will go to the cluster level arbiter, which will issue a bus granted signal. In the cycle after a bus granted, the core will put the request on the cluster level bus and the request will be issued to the cluster level instruction cache. After some number of cycles, depending on whether the request was a hit on the cluster level cache, the cluster level cache will issue a response onto the cluster level snoop response bus. At this point the L1I will be able to snoop the response; and if the response is valid and the response was intended for this core, the cache will recognize this response as a valid response for the request it issued. The L1I will respond to the core in the same cycle in which a valid response was given, without waiting for a fill. This is an improvement over the traditional read-only-from-cache scheme, and in this case, it saves

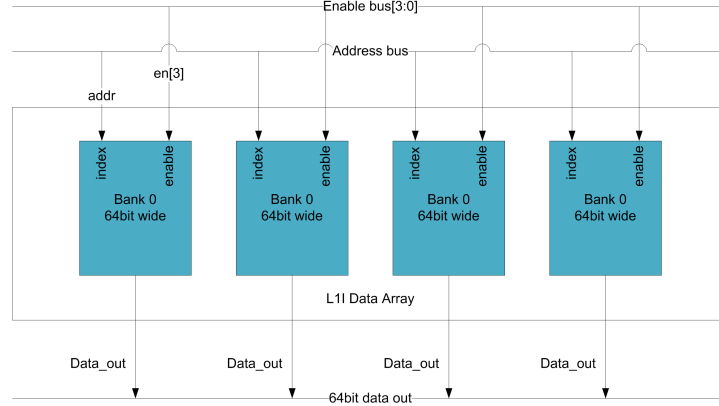


Figure 2.1: L1I data array

us two cycles. This optimization is achieved by multiplexing the response from the cache and the cluster level response bus and sending the output to the core. As expected, we will choose the response from the SRAM on a hit and the response from the cluster level response bus on a valid response from the cluster level cache. This optimization saves us two cycles of execution due to the synchronous SRAM, which requires a cycle to latch the request and a cycle to resolve it.

### 2.3.1 SRAM choices

For the L1 instruction cache I decided to use single ported SRAMs. Both the tag and the data are stored in an SRAM. The tag SRAM will vary in bit width depending on the number of index bits. To calculate the tag bits I use the following calculation:  $TAG\_BITS = 32 - INDEX\_BITS - OFFSET - 2$ . I subtract two since our architecture is word aligned. The L1I cache line is 256 bits wide. To implement this data line in the cache I have chosen to use 4 SRAM banks with width of 64 bits. Figure 2.1 illustrates the SRAM array organization. The reason for this design decision comes from the core fetch stage requirement. The core attempts to fetch two instructions every cycle, which is 64 bits total in our 32 bit architecture. With this implementation I optimize the L1I cache to only activate a single bank out of the 4 on a cache read. This optimization can cut the dynamic power of the data line SRAM by a factor of four. However, on a cache fill we are still forced to activate all the SRAM arrays. This optimization requires a single 2-to-4 decoder. I did

not study the exact effect of this optimization. The instruction fetch requires no support for stores, which is the main reason for my decision to use a single ported SRAM for the L1I. One might argue the possible benefit of using dual ported SRAM to support same cycle fill and read. However, I claim that our code is very regular and most reads will be sequential, which will result in same row read/write which our SRAMs do not support, resulting in a stall. The extra area and complexity of the SRAM will not give us any significant performance gains.

### 2.3.2 Pipelined cache read

An important optimization I have implemented in the L1I cache is the pipelined read. As seen in Figure 2.2, the fetch unit actually comprises two stages. The first stage is address calculation and logic to support branches and fetch stall. In terms of hardware the first stage consists of an adder and a multiplexer. The first stage is connected to the L1I cache input. The second stage will stall until a valid cache response arrives. To support this feature we are also required to latch the request internally in the cache so that the first fetch stage can change the request on the core-to-cache interface. This feature allows us to send a new request in the same cycle the previous request is resolved, resulting in a single cycle read.

### 2.3.3 Branch handling

As mentioned before, we have decided to design a simple area-efficient core, so we have decided to eliminate all branch prediction logic. We have also decided to prevent speculative fetch, since its efficiency without branch prediction logic will be low. These decisions are reflected in our L1I design in that once we detect a branch instruction in the decode logic we will stall the fetch unit until the branch is resolved, and resume fetch from the branch address in the same cycle it is available.

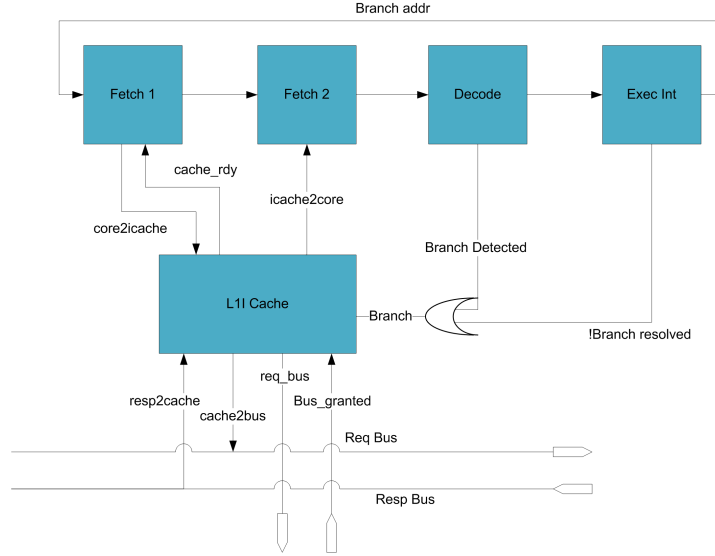


Figure 2.2: L1I block diagram

### 2.3.4 L1I block diagram

Figure 2.2 shows the interaction between the fetch stage in the core and the L1I cache. The first stage sends the request to the cache only if the cache indicates that it is available to accept a new request and the second stage consumes the response. This pipeline allows us to resolve one request per cycle even with a synchronous SRAM in the cache. Also, Figure 2.2 illustrates the interaction between the decode and execution stages and the fetch stage. The decode stage can detect that a branch is in the pipeline, causing the fetch to stall. The execution stage indicates when a branch is resolved and provides the branch target address for the fetch unit.

#### Internals

Figure 2.3 shows the internals of the L1I cache. The cache comprises three main components: controller, direct mapped cache and output logic. The diagram is not meant to illustrate all the gates, but is a general picture of the organization of the cache. Inputs from the core and the storage arrays will change the controller state. The core compare block will signal the controller that a valid cluster response was received. The cache will send a ready signal if the cache does not have any request in-flight.

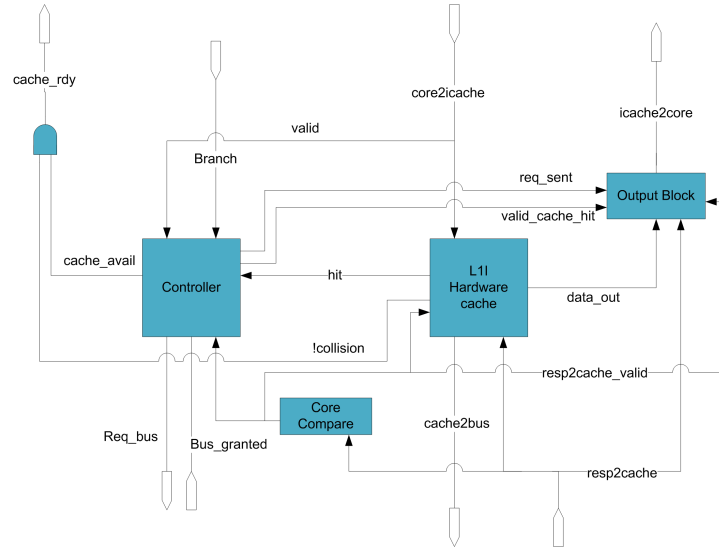


Figure 2.3: L1I block diagram detail

### Direct mapped cache

Figure 2.4 shows the internals of the direct mapped cache block. The request from the core will be latched into a register internal to the direct mapped cache. The inputs to the hardware are the request from the core, and the response from memory. The two are multiplexed as input to the address of the SRAM arrays. The SRAM arrays can be activated both by read from the core and write from the fill. In case of a load, the 2-to-4 decoder will activate only a single bank in the data array based on the block offset, as described in Section 2.3.1.

### Output Block

Figure 2.5 illustrates the output logic that selects between the response from memory and response from the cache. The motivation behind it was described in Section 2.3

### 2.3.5 L1I state diagram

Figure 2.6 shows state machine described in the cache design section. It is worth mentioning that although the read is pipelined there is a centralized controller that controls the cache. The specifics of the pipeline design were



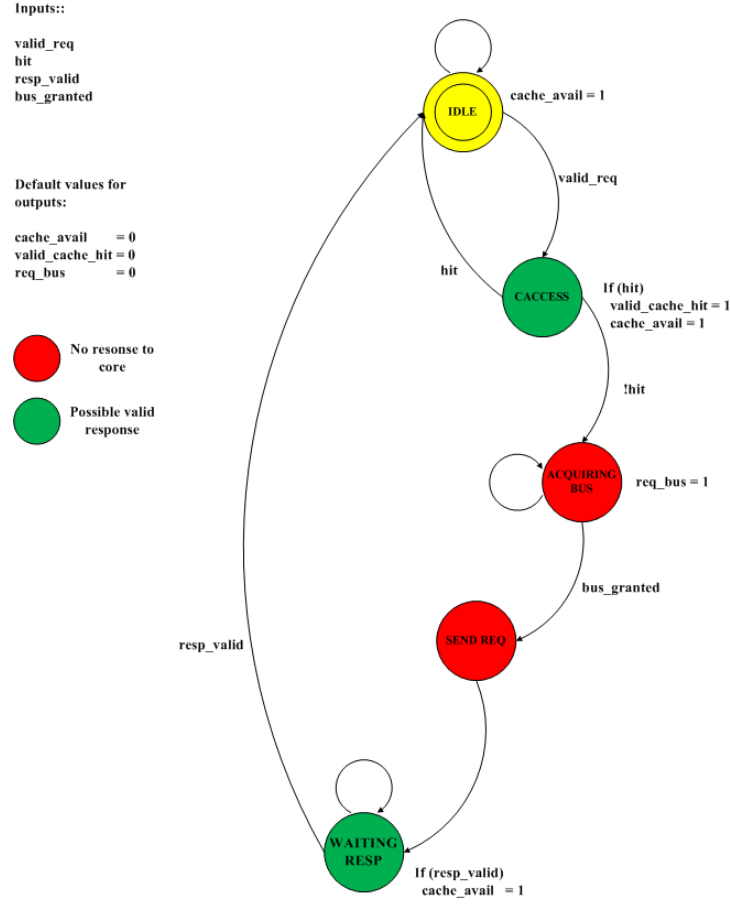


Figure 2.6: L1I FSM

described in Section 2.3.2.

## FSM States

*IDLE* is the reset stage, where no request is being handled. If a new valid request is issued, the cache will transition to the *CACCESS* stage. If there is an unresolved branch in the pipeline, the cache remains in this state.

*CACCESS* is the stage where the SRAMs are checked for a tag hit and the data arrays are read. If the request is a hit in the cache, and there is no new incoming request, the cache will transition back to the *IDLE* stage. If the request was a hit and there is a new valid request, the cache will remain in the *CACCESS* stage. If the request missed in the cache, the cache will transition to the *ACQUIRING\_BUS* stage.



*AQUIRING\_BUS* is the stage in which the cache will assert the req.bus signal and wait for a grant from the bus controller. The cache can be in this stage 1-9 cycles depending on the contention on the bus. Once the bus is acquired, the cache will transition to the *SEND\_REQ*.

*SEND\_REQ* is the stage where the cache will put the request on the request bus. This is guaranteed to take a single cycle since the bus will not be granted unless the cluster cache can guarantee to accept the core request. I separate the bus acquiring stage and the request sending stage to prevent a critical cycle.

*WAITING\_RESP* is the stage where the cache will stay until a response from the cluster cache will be received. Once a valid response has been received, the cache will transition to the IDLE stage. There is no transition to the *CACCESS* stage since it takes two cycles to store to the synchronous SRAM arrays.

#### FSM Inputs

**valid\_req** input signal indicates whether there is a new valid request.

**hit** input signal indicates a cache hit.

**resp\_valid** input signal indicates that a valid response from the L2 was received.

**bus\_granted** input signal indicates that the bus was granted to the core.

#### FSM Outputs

**cache\_avail** output signal will indicate if the cache is ready to accept a new request. Default value is 0.

**valid\_cache\_hit** output signal indicates if a cache hit happened in the *CACCESS* stage. Default value is 0.

**req\_bus** output signal will notify the bus controller that the core is requesting the bus. Default value is 0.

# CHAPTER 3

## L1 DATA CACHE

The effort of designing a data cache is considerably larger than the instruction cache. The data cache, unlike the instruction cache, needs to support stores to memory. Although it is a simple concept, it is one of the main sources of the design complexity and design choices of the data cache. The write feature requires a designer to consider coherence as discussed in [9] and consistency as discussed in [10]. Another feature that makes the data cache distinct from the instruction cache is that it can have several independent streams of data access. As a result, the execution time will be reduced if the data cache can resolve several cache misses simultaneously. This chapter will discuss the L1 data cache design, explain the design tradeoffs and elaborate on the chosen design and the implementation details.

### 3.1 L1D Write Policy

Jouppi [21] discuss in depth the different write policies and explain the trade-offs each presents. I will discuss briefly the different considerations when choosing a write policy and later explain the policy I chose to implement. The write policy has significant impact on the hardware and on the overall performance. However, there is no correct choice; there is only the choice that makes the most sense for a set of applications.

#### 3.1.1 Write-miss policy

Write-miss policy refers to the cache operation in case the write results in a cache miss. There are two actions that the cache can perform on a write miss. The first is fetch, which we refer to as fetch-on-write, refers to the allocation of the entire line on a write miss. The second is allocation of the

written data, which we will refer to as write-allocation. Jouppi studies in detail the policies that result from the different combinations of the above actions on a miss and show that each policy can outperform the others given the right application.

### 3.1.2 Write-hit policy

The designer also needs to decide on the policy a cache will take when a write hits in the cache. The two policies are write-through and write-back. When using the write-through policy, every write will always propagate to a higher level storage. With write-back policy, only the local copy of the data will be modified on stores and the higher level storage will see modified data only on an explicit data copy or on a cache line eviction. Not all write-miss policies make sense with both write-hit policies. For example, having a no-write-allocate policy does not make sense when combined with a write-back policy.

### 3.1.3 The chosen write policy - Write-around

I have chosen to implement the write-around policy. The write-around policy is a no-fetch-on-write and no-write-allocate, which means a write miss will not result on a line fetch or allocation of the written word. The write-around only makes sense when combined with the write-through policy. On a cache hit the write will modify the local copy, but will also propagate the data to the next level of storage. Write-through policy in the first level cache was chosen due to several advantages:

- Bandwidth requirements can be satisfied for most applications. However, even when contention for cluster cache access is high, the effect is diminished by longer latencies in the system, such as main memory access latency.
- Write-through combined with inclusive cluster cache allows the cluster cache to have a coherent view of the entire cluster memory, which allows it to be a point of synchronization. Simply invalidating a line in the L1D cache will result in cluster level synchronization.

- Load operations will have shorter latencies since there is no need to support extra cycles for eviction in case of a conflict.
- The policy is easier to implement and requires less hardware, especially with a nonblocking cache design.

As a result of the listed advantages, it is a common design choice in parallel systems for the L1 data cache. Commercial designs, Sun's Niagara [2] and IBM's Power5 [22], and the academic design Hydra [4], also made similar design choices.

## 3.2 L1D Design

In addition to the write policy, the cache designer needs to decide on the actual design of the cache. There are many design features when it comes to cache design. Investigating all the different features and their tradeoffs is beyond the scope of the work. Instead I will focus on some key features that potentially have a large impact on the design. Future work can reuse the methodology and design to explore further design features.

Some different design features and tradeoffs that exists are:

**Cache Associativity** What is the optimal associativity?

**Cache Size** What is the optimal cache size for the design?

**Load/Store Queue** Allows the core to load data from a local store.

**Blocking vs. Nonblocking Designs** Enables the cache to resolve several misses simultaneously.

**Write-Back Buffer** Allows coalescing of write-backs, reducing bandwidth requirement.

The nature of the application the Rigel architecture targets does not require a highly associative cache; hence, I have decided to implement a direct mapped cache. In my work I will only explore the cache size and the non-blocking features of the cache design space. In the result section I will explore the effect of varying the L1D cache size and the blocking vs. nonblocking

designs and show the impact on the overall performance, power and area of the system.

To start I will describe the design features that are common to all the different design variations:

- All of the designs will share the same direct mapped cache design, with identical SRAM array organization. When varying the size I will only modify the number of lines in the cache.
- In all cases, stores will proceed once they have been sent to the cluster cache. The core should not wait for a response; neither will a response come for a store due to the chosen write-around policy.
- The stall signal for the first memory stage in the core will depend on the cache ready signal to accept a new request. The cache ready signal will consist of two components: whether there is a request already in-flight and whether the new request has a collision with the previous memory operation which prevents it from being pipelined. A collision happens when a read and a write target the same line.

### 3.2.1 SRAM choices

For the L1 data cache I decided to use two ported SRAM with one read and one write port. Both the tag and the data are stored in SRAMs. The tag SRAM will vary in bit width depending on the amount of index bits. To calculate the tag bits I use the following calculation:  $TAG\_BITS = 32 - INDEX\_BITS - OFFSET - 2$ . I subtract two since our architecture is word aligned. The L1D cache line is 256 bits wide. To implement this data line in hardware I have chosen to use 8 SRAM banks with width of 32 bits since each load and store will be exactly 32 bit wide. With this implementation I optimize the L1D cache to only activate a single bank out of the eight on loads and stores. This optimization can cut the dynamic power of the data line SRAM by a factor of eight. However, on a cache fill we are still forced to activate the entire data cache SRAM. Having a dual ported SRAM also allows us same cycle read/write with proper pipeline support, as will be described in the next section.

### 3.2.2 Pipelined cache read/write

Similarly to the L1I, the L1D also supports pipelined read, allowing the synchronous SRAM to emulate a single cycle read. However, in contrast to the L1I, the data cache has a requirement to support stores. Store will take two cycles in the current implementation, due to the nature of the synchronous SRAMs. It takes one cycle to read the tag and a second cycle to check for a hit and perform the cache store. To hide that latency I have added support to pipelined read/write. As long as there is no address conflict, the design can support same cycle read/write. This is the main motivation behind a dual ported SRAM. As explained earlier the cache is write-through, which means the cache will only complete the store when it is sent to the next level of storage. However, we can benefit from this optimization in a nonblocking design, since the frontend that accepts the stores is disjoint from the backend which sends the requests to the cluster cache.

### 3.2.3 Blocking design

The core and blocking data cache are connected by two separate unidirectional buses. The request bus consists of a 32 bit data bus, 32 bit address bus, 5 bit request message bus, and a single bit valid bit. The L1D response bus consists of a 32 bit address, 32 bit data, 5 bit response message bus, and a single valid bit. The unidirectional interconnect is simpler to implement and allows us to issue a request and receive a response at the same cycle.

If the cache is available and the core has a new request, the core will put a valid request on the request bus. The cache will decode the request and output control signals to be used by the controller. In the next cycle the request and the decode packet will be latched into the cache and the SRAM latch and be checked for a cache hit. If a cache hit occurred the cache will respond to the core with valid data. In case of a store hit the cache will write the data to the data array. If a request missed in the cache or it is a store operation, the cache will assert the request bus signal, indicating it needs to issue a request to the next level cache. The request will go to the bus controller which will issue a bus granted signal based on the arbitration policy. The cycle after a bus granted the core will put the request on the cluster level bus and the request will be issued to the cluster cache. If the

request is a store, at this point the cache will respond to the core with a valid response. After some number of cycles, depending on whether the request was a hit in the cluster cache or not, the cluster level cache will issue a response onto the cluster level snoop response bus. At this point the cache will be able to snoop the response, and if the response is valid and the response was intended for this core the cache will recognize this response as a valid response for the request it issued. The cache will respond to the core in the same cycle a valid cluster cache response was given, without waiting for a fill. This is an improvement over the naive implementation which would read only from the L1D, and it saves us two cycles of execution. This optimization is achieved by multiplexing the response from the cache and the cluster level response bus and sending the output to the core. As expected, we will choose the response from the SRAM on a hit and the response from the cluster level response bus on a valid response from the cluster level cache. This allows us to save two cycles of execution due to the synchronous SRAM, which requires a cycle to latch the request and a cycle to resolve it. We avoid conflicts between core loads and stores and cache fills by having the collision logic, which stalls the core in case of read/write to the same line in the same cycle. We prevent conflicts between stores which already completed the tag lookup and fills by giving priority to fills over the write port.

### Blocking Design Block Diagram

Figure 3.1 shows the interaction between the memory unit in the core and the L1D cache. The first stage sends the request to the cache only if the cache indicates that it is available to accept a new request and the third stage consumes the response. This pipeline allows us to resolve one request per cycle even with a synchronous SRAM in the cache.

Figure 3.2 shows the internals of the L1D cache. The cache comprises four main components: Controller, request decoder, direct mapped cache and some output logic. The diagram is not meant to illustrate all the gates, but is a general picture of the organization of the cache. Inputs from the core, request decoder and the direct mapped cache will change the controller state. The request decoder will decompose the request to control flags such as the `is_store` control flag. The core compare block will signal the controller that a valid response was received. The cache will send a ready signal if the

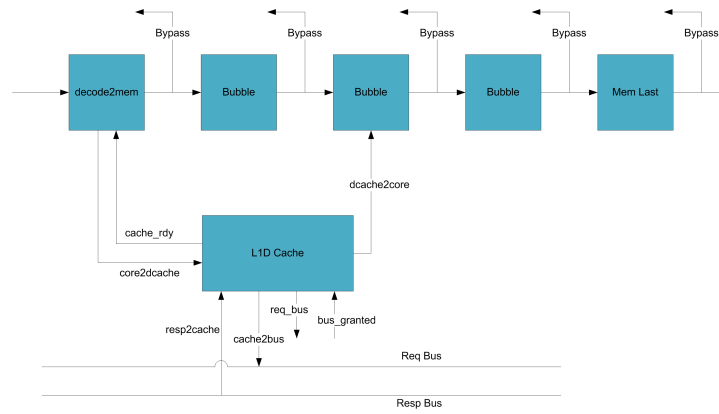


Figure 3.1: Blocking L1D block diagram

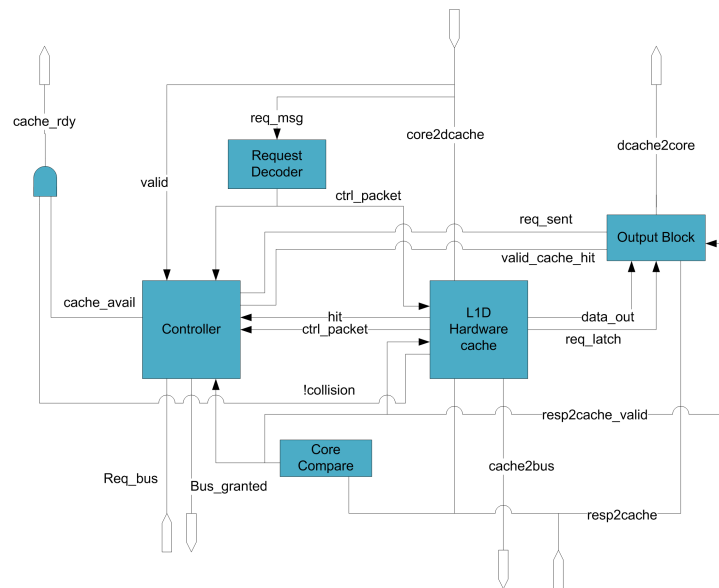


Figure 3.2: Blocking L1D internals



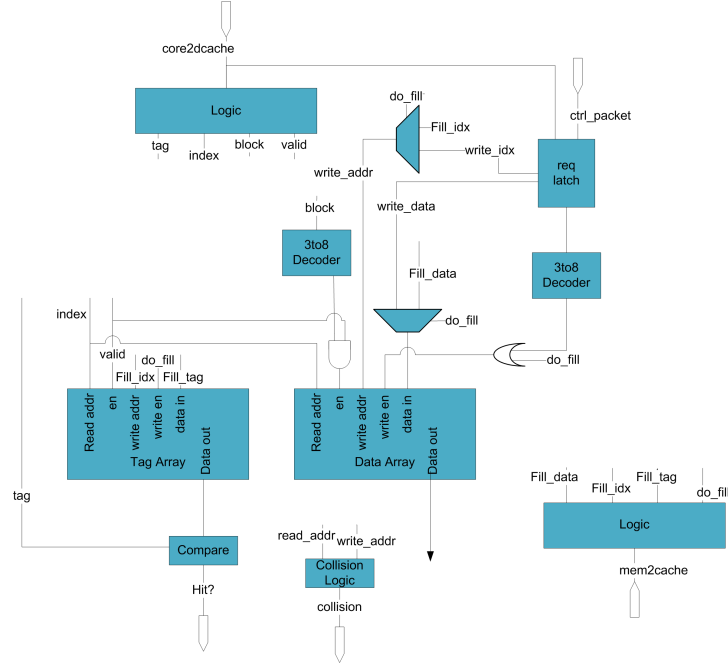


Figure 3.3: Direct mapped cache diagram

cache does not have any request in-flight.

Figure 3.3 shows the internals of the direct mapped cache block. The request from the core combined with the controls flags from the request decoder will be latched into a register internal to the direct mapped cache. The inputs to the hardware are the request from the core, and the response from memory. The core request will be an input to the read port of SRAM arrays. The latched request and fill are multiplexed as input to the write port. In case of a load, a 3-to-8 decoder will activate only a single bank in the data array based on the block offset. A separate decode will do the same for a store. In case of a fill the entire SRAM array will be activated. Figure 3.3 shows that the fill and store are multiplexed for the input to the SRAM arrays write port, where the priority is given to the fill. It is a key implementation detail to prevent store/fill conflicts.

Figure 3.4 illustrates the output logic that selects between the response from memory and response from the cache. The output block will also respond to the core if a store request was sent to the cluster cache.

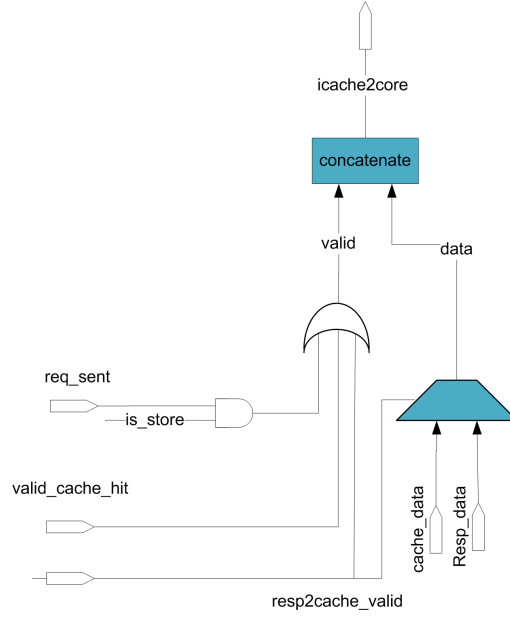


Figure 3.4: Blocking L1D output block

### Blocking design FSM

Figure 3.5 shows the state machine described in the cache design section. The blocking cache state machine is based on the instruction cache FSM in Section 2.3.5 with a couple of differences: The additional input signal `is_store` indicates if the request is a store operation. The cache will send a valid response to the core in the *SEND\_REQ* stage if the request is a store.

### 3.2.4 Nonblocking design

The nonblocking L1D interface adds several components on top of the blocking design interface. The cache response bus adds a two bit component for an index to response array. As seen in Figure 3.6 we add two additional unidirectional buses to interface the core with a response array. The last memory stage request bus will use the index supplied in the second stage of memory to index into the response array. The response array will respond with the data and a response message when available.

In case of a cache hit, the cache will operate similarly to the blocking design. In case of a cache miss, the core will not block the request until resolved. Instead, the missed request will put the request on the Miss Status Handling Register (MSHR) array, and let the next memory request access

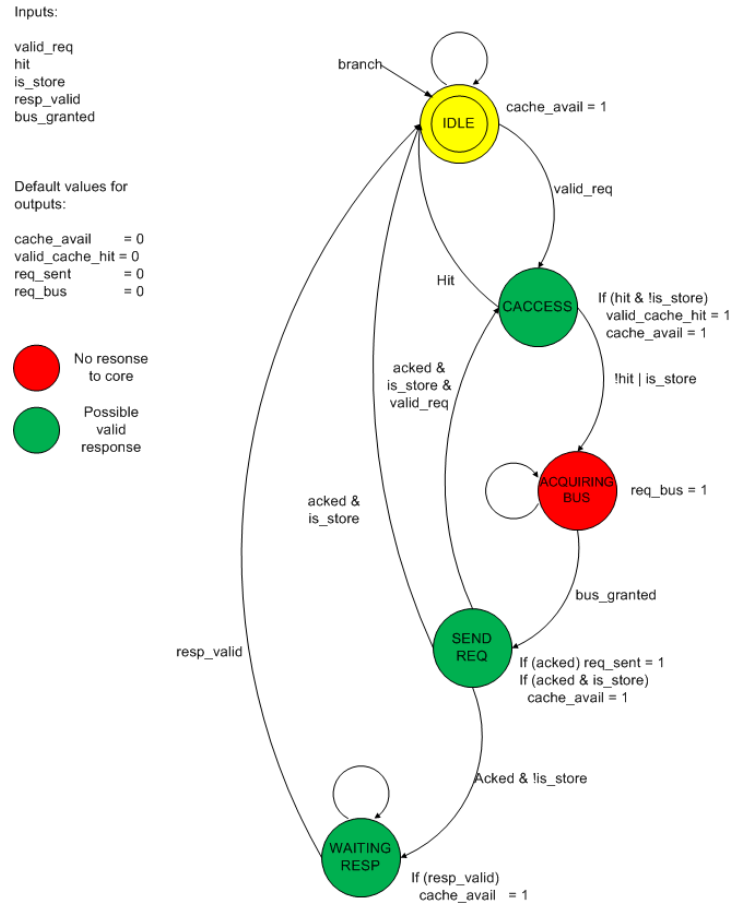


Figure 3.5: Blocking L1D FSM

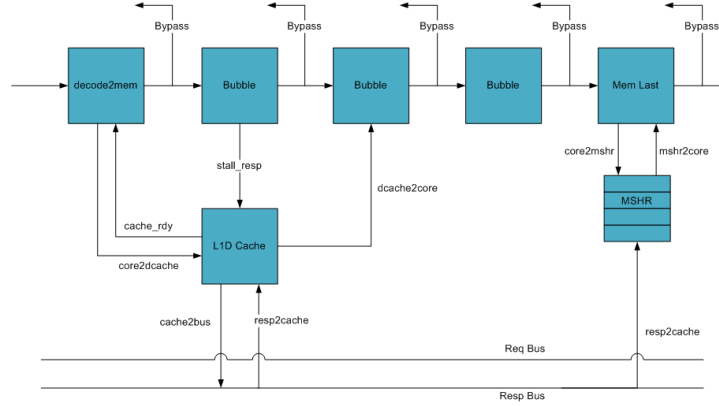


Figure 3.6: Nonblocking L1D block diagram

the cache. The miss will stall at the last stage of the memory unit it gets the response from a separate smaller structure. The MSHR array will drain in order of arrival, independently from the core operation, and collect the responses from the cluster response bus into the response array. This enables multiple outstanding misses pending simultaneously. Also, this allows the other execution units to utilize their pipe stages more efficiently. For all stores, the cache will not block for the write-through. The nonblocking cache can potentially increase the crossbar performance over the blocking design by hiding the latency penalty. It stalls the memory response only at the last stage of memory, and allows issuing one request per cycle to the interconnect.

### The MSHR Structure

The main structure that enabled us to have a nonblocking cache design is the Miss Status Handling Register (MSHR) array, which serves two main purposes: It buffers the requests and allows us to perform a fully associative lookup in the pending requests. Figure 3.7 illustrates the hardware of the MSHR array. To enable a fully associative lookup, we have a comparator per entry. The associative lookup enables us to determine if a load to a certain line is already pending. To ease consistency issues and for hardware simplicity, the cache will stall in case a new request is trying to access a cache line that is pending a response. This feature puts more strict requirements on the parallelism of the nonblocking cache. To exploit MLP with this design, we need a stream of memory requests to different cache lines. In addition, we need a FIFO controller to accept and drain new requests in order of arrival.

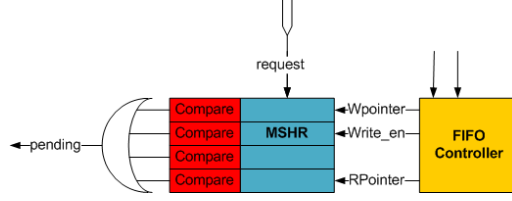


Figure 3.7: MSHR diagram

I implemented the FIFO controller with a Synopsys DesignWare IP.

Each MSHR entry will contain several pieces of information:

- Address of the memory request
- Write data, in case the request is a store
- Request message
- The state of the MSHR

The MSHR can be in three states: EMPTY, PENDING, REQ\_SENT. The PENDING state indicates that the request was not sent to the clusters cache. The REQ\_SENT indicates that the request was sent to the bus; only load request will transition to the state to support the load pending check. Once a response from the cluster cache comes, the cache will transition back to the EMPTY state.

### Nonblocking Design Block Diagram

Figure 3.8 illustrates the additional signals and hardware blocks required to support the nonblocking cache. The controller is broken down into two independent controllers. The MSHR array will be controlled by the two controllers and bus responses. The cache\_rdy signal has an added component that comes from the MSHR structure. The MSHR will send a stall in case a load to the line is already pending, or there is no empty entry in the array. The output block will not multiplex the cache response with the bus response but simply pipe through the cache response.

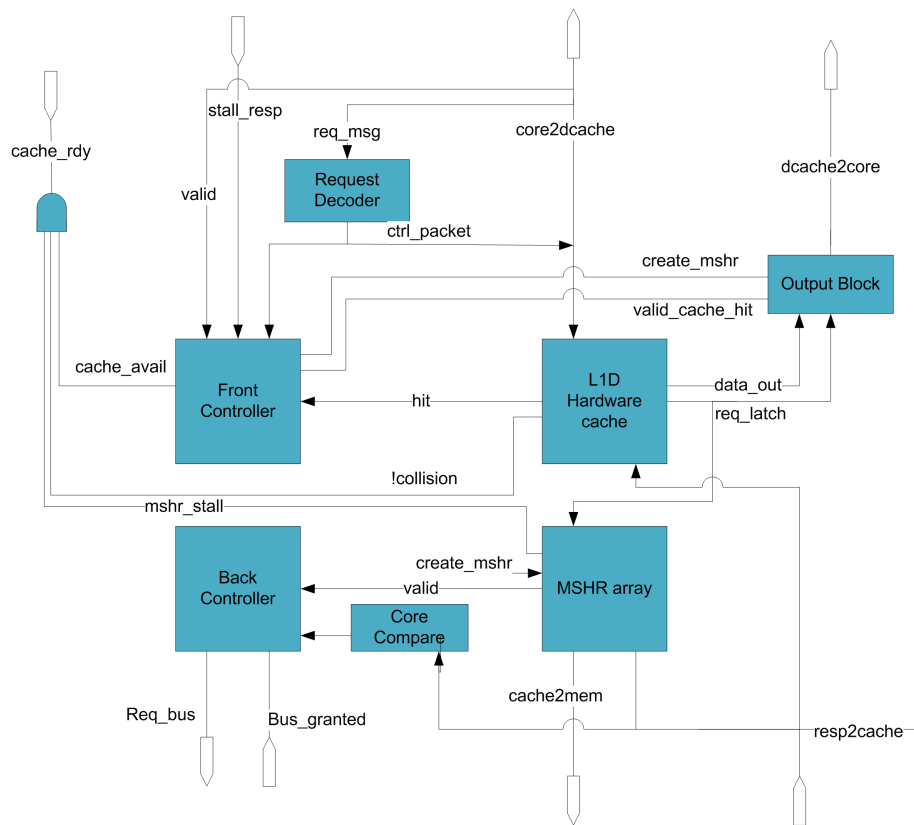


Figure 3.8: Nonblocking L1D internals

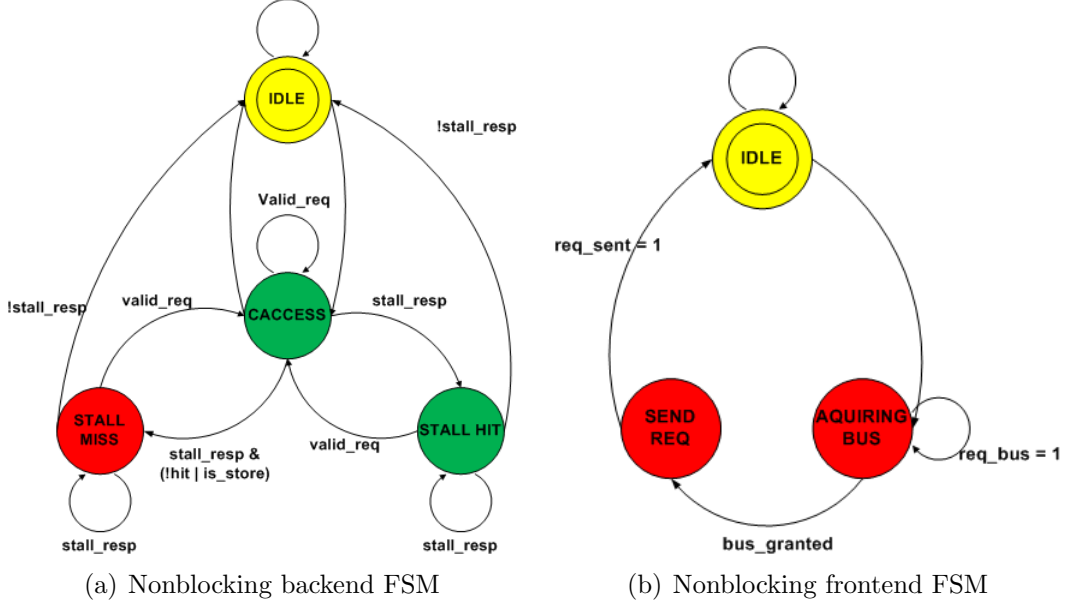


Figure 3.9: Nonblocking L1D FSMs

### Nonblocking Design State Diagram

The nonblocking design will have two state machines: frontend Figure 3.9(a) and backend Figure 3.9(b). The frontend controller is responsible for cache access, stalling the response in case of a core stall and delivering the request to the MSHR. The backend controller is responsible for draining the request from the MSHR array in order of arrival. If requests are pending on the MSHR array, the backend controller will request the cluster cache request bus and send the request.

In addition to the blocking L1D design, the nonblocking design introduces the following new states:

*STALL\_MISS* if the core request to stall the cache response and the request was a cache miss.

*STALL\_HIT* if the core requests to stall the cache response and the request was a cache hit.

### FSM Inputs/Outputs

The nonblocking design shares the same inputs and outputs with the blocking design, which the addition of single input signal. The input signal *stall\_resp*

will stall the cache lookup result as long as the signal is asserted by the core. This signal exists because the core is required to get the MSHR index of the current request in case of a miss. If the request does not stall, but the core does, due to a pipeline conflict, the MSHR index will be incremented, and the core will receive the wrong MSHR index.

### 3.3 Cluster Interconnect Impact

The changes to the L1D required us to switch from a bus architecture to a crossbar architecture:

- Eliminate the hardware for the direct mapped cache block.
- Eliminate the need for the *CCACHE* stage in the state machine, both in the blocking design FSM and the frontend of the nonblocking design.
- Output block - Responses can only come from the cluster response bus, so the output logic is simplified to forwarding the cluster cache response to the core.
- Eliminate the comparators for each MSHR entry, since each request needs to go to the cluster cache even if its a duplicate.



# CHAPTER 4

## CLUSTER INTERCONNECT AND CONTROL

### 4.1 Cluster Interconnect

#### 4.1.1 The tradeoffs

As mentioned before, I will study two cluster interconnects: a bus and a crossbar. First, I will discuss the tradeoffs between the two designs.

The following are the key tradeoffs:

**Design Complexity** The bus interconnect has the advantage of being a much simpler design, which will be easier to lay out. The crossbar has many more wires, which will be more challenging to work with in the place and route stage.

**L2 Access Frequency** The L2 is a much larger cache than the privatized core L1; hence, a L2 access is much more costly. We want to minimize the access frequency to the L2. With a bus interconnect, the L1 serves as a filter to minimize L2 access. With crossbar interconnect, even though we do access the cluster cache constantly, we only access the appropriate bank.

**Interconnect Contention** Application with bad locality can create high contention for the L2 interconnect. The bus interface can potentially be the performance bottleneck in the system.

**Memory Operation Latency** With the bus, a memory operation can have a cycle hit due to the existence of a privatized L1. However, with a crossbar the memory latency is at least four cycles. The programmer and the compiler need to accommodate for that fact and hide the latency by rearranging code and unrolling loops.

**Area/Power Trade-Offs** With a bus interconnect there is a privatized L1 which adds power and area. The crossbar interconnect does not have a privatized L1; however, there is logic duplication due to banking. Also, more power and possibly area (depending on the metal layer used) are consumed by the crossbar wiring.

**Multithreading Tolerance** The more multithreaded the core is, the more conflicts the privatized L1 is going to incur. Multithreading provides several independent memory streams which will perform poorly with a simple L1. To prevent conflict misses, there is a need to build a highly associative complex L1 in case of a bus interconnect. With a crossbar, there is no L1, so this problem simply does not exist.

**Coherence and Consistency** Having a privatized L1 complicates coherence and consistency models, since the local view of data needs to be kept coherent with the global view. Since I have chosen to take a software coherence and processor consistency approach, there will be no adverse effects. However, if in the future we want to modify the design to hardware coherence or a more strict consistency model, it will have an adverse effects on design complexity and performance. However, a crossbar interconnect avoids these problems altogether by having a single shared cluster cache.

#### 4.1.2 L1D to L2D - Bus architecture

The first interconnect design I will discuss is the bus interconnect. The bus interconnect is comprised of two unidirectional buses: request bus and response bus. The request bus passes requests from the core to the cluster cache. Figure 4.1 illustrates the request bus. The cores are connected to the cluster request bus with tri-state buffers controlling the output. Figure 4.2 illustrates the core-to-bus control connectivity. The cores request the bus and the arbiter responds with a grant signal that will enable the appropriate tri-state buffer. Figure 4.3 shows the response bus. The cores will snoop the response bus for a response from the cluster cache. Only the cluster cache controls the response bus; hence, there is no need for tri-state buffers.

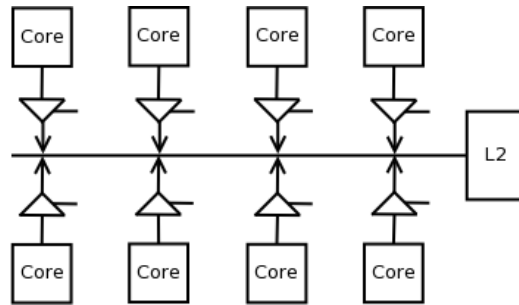


Figure 4.1: Core to L2

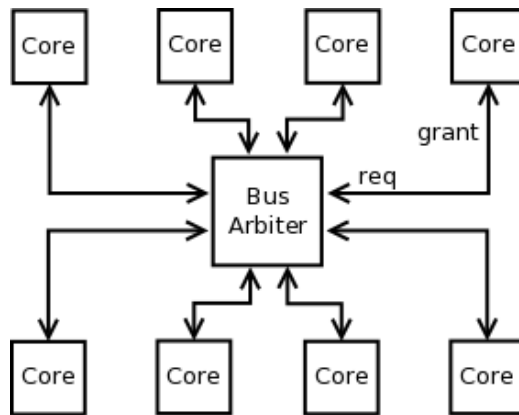


Figure 4.2: Core to L2 arbiter

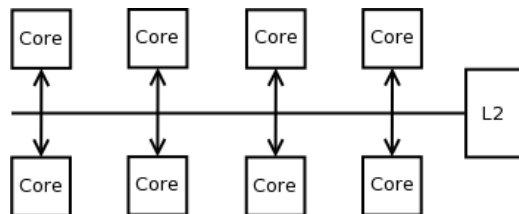


Figure 4.3: L2 to core

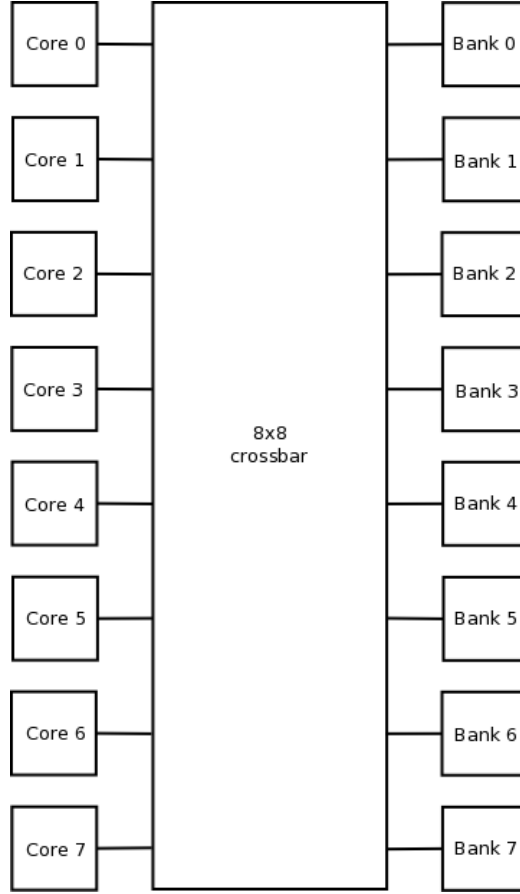


Figure 4.4: Crossbar interconnect

### 4.1.3 L1D to L2D - Crossbar architecture

The crossbar interconnect connects all cores with all cluster cache banks. Figure 4.4 shows the organization of core and banks in the cluster. Figure 4.5(b) is an example for  $2 \times 3$  crossbar implementation. The crossbar interconnect comprises two unidirectional crossbars. The request crossbar is implemented with a per core output bus and a per bank input bus. Each input bus has a controller to arbitrate between several incoming requests, as seen in Figure 4.5(a). The respond crossbar is implemented with a per core input bus and a per bank output bus. Each core input bus will be controlled in case several responses target the same core.

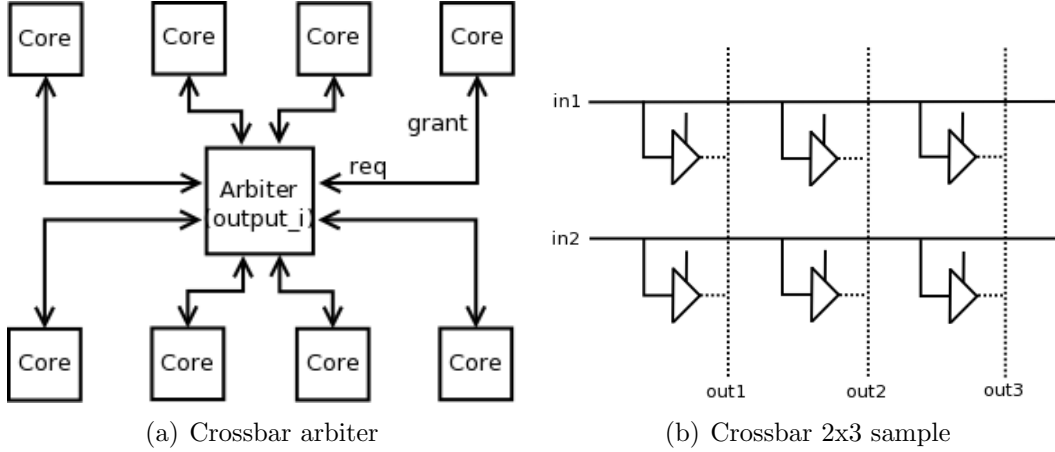


Figure 4.5: Crossbar implementation

#### 4.1.4 L1I to L2I

To focus on the data part of our application, I separated the instruction and data cluster interface. As discussed previously, the core will always have a privatized L1I connected over a bus interface to a cluster level cache.

## 4.2 Interconnect Control

The main unit of control for both interconnects is the bus controller illustrated in Figure 4.6(a). The bus controller comprises a state machine shown in detail in Figure 4.6(b), an arbiter and a latch used as an input for the arbiter. The bus has a single instance of the bus controller, controlling the request bus. The crossbar has eight instances for request bus and eight instances for respond bus.

### 4.2.1 Arbiter design

The arbiter is the hardware controlling the bus access policy. The arbiter provides the bus controller a one-hot, eight bit vector, which tells the controller which core gets the bus grant. The arbitration policy is a circulating priority based on the last grant. That means that the core currently holding the bus was the highest priority on the last arbitration cycle, and on next arbitration cycle it will become the lowest priority. The priority is descending

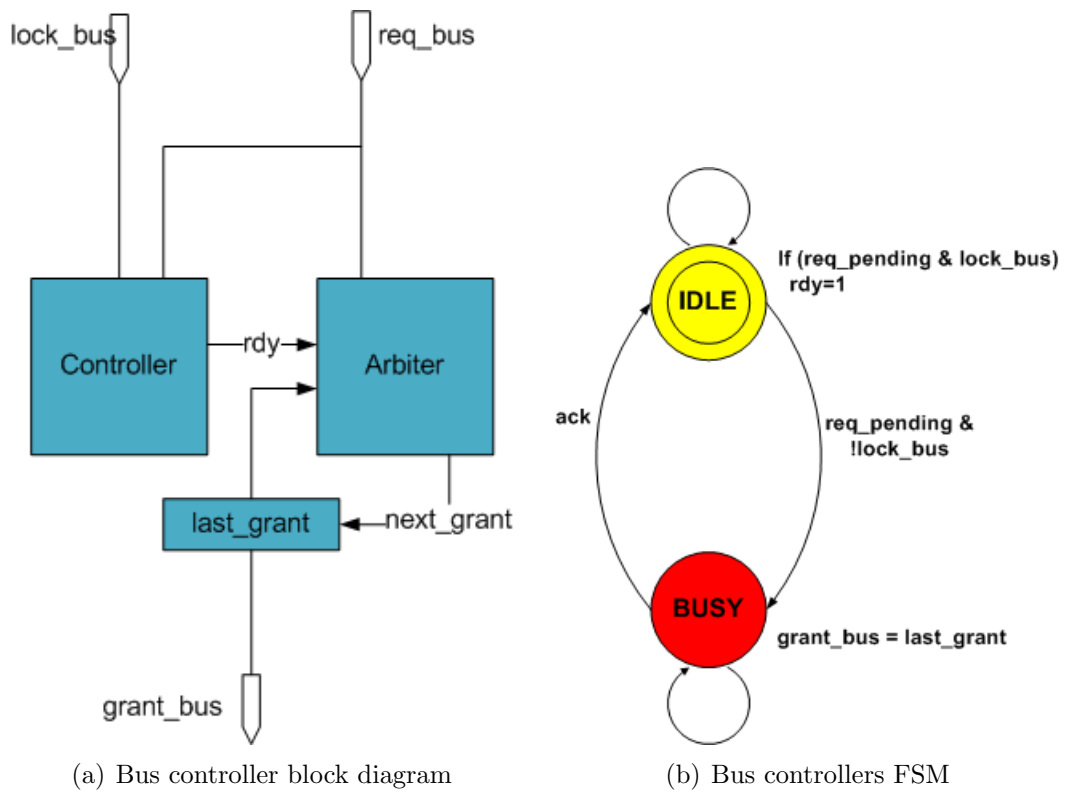


Figure 4.6: Bus controller

from the core following the last grant, all the way around to the core that had the last grant. To implement this arbitration policy we require eight 3-to-8 decoders, one decoder per priority setup. This policy requires more costly hardware than a simple static priority round robin, which requires only a single decoder; however, it prevents a starvation situation.

# CHAPTER 5

## CLUSTER CACHE

In this chapter I will discuss the implementation of the cluster cache. The cluster is the basic building block of the Rigel architecture. It is important to carefully decide on the cluster architecture. The cluster design will be duplicated 128 times to create the 1024-core Rigel chip. Needless to say, with the cluster design, the error in power and area measurements will be multiplied by the same factor.

A major contributor to the power and area of the cluster is the cluster cache. As a result, to get accurate estimation of area and power for the entire cluster, the cluster cache needs to be part of the studied RTL implementation. Even if the cluster cache is simple and the RTL model incomplete, it brings us a step closer to post-silicon power and area results.

### 5.1 Splitting Instruction and Data

In order to focus on the data cache design tradeoffs, I made the instruction and data cluster level interaction completely disjoint, essentially creating separate cluster level cache for instruction and data. This design decision was made due to time constraints and the minor impact of having a unified cluster cache. Only the cluster level data cache will be studied, implemented in RTL, simulated and synthesized. Hence, when I refer to cluster cache in my evaluation it only concerns the data streams of the application. The instruction memory model will always have a privatized L1 with a bus interface to the instruction cluster cache. In the L1 instruction cache I explained the reasoning for having a privatized L1. Implementing the mixed cache will not affect our results significantly since the target applications are regular and consist mainly of an inner loop which can be completely contained in the L1. Hence, the effect of instruction traffic on the interconnect and cluster cache



will be minimal.

## 5.2 Cache Design - Reuse of the L1D Design

Designing a cluster cache can be a very complex task. A fully featured clustered cache should be able to combine and reorder memory requests and to support advance operation such as atomics. However, implementing a fully featured cluster cache is beyond the scope of my work. As a result, cluster cache work will be based on the L1D design specification described in the previous section. Some design decision were ported to the cluster cache purely due to time constraints; however, I believe this simple cluster cache design can give us good relative estimation for performance, power and area when studying the design tradeoffs. Adding the advanced features discussed above will provide additional design points and will not change the relative results. I do not expect that having requests combined and reordered will affect the results significantly in most of my applications since they are all working on disjoint data in a very regular pattern.

Some prevalent design decisions that should be reconsidered instead of the naive port from the L1 are the write policy and the cache associativity. The cluster cache, similarly to the L1 data cache, will use a write-around policy with write-through. Although stores are not major components of my application and are not blocking similarly to the L1D, this decision should be revisited in the future. In my experiments the write policy did not have an effect on performance since I was not simulating global interconnect contention. However, when considering the full Rigel chip, having 128 cluster caches with a write-around policy can create global interconnect contention, which can potentially be a performance inhibitor. In addition, the L1D can use a simple direct mapped cache since it does not need to support many independent data streams; however, the cluster cache needs to support data streams from all eight cores in the cluster, making it much more susceptible to conflict misses. As a result, a higher associativity can be very beneficial.

The cluster cache implementation is identical to the L1 data cache, with the exception of the following key differences:

**In/Out FIFOs** to enable request and respond buffering, preventing bus stalls. This allows the core to send the request even when the cluster

cache is busy and continue with independent execution.

**Stores** will not generate a valid response since we do not fetch on write a write miss.

**Cache stall logic** will stall cache responses in case of simultaneous valid response from the cache and memory.

**MSHR structure** will store the MSHR index in the L1 nonblocking cache.

This is required so that the cluster cache response can contain the originating L1 MSHR index.

**Output width** is 256 bit, in contrast to the 32 bit of the L1D.

The internal design, the state machines and direct mapped cache design are all ported from the L1 data cache. Also, the cluster cache will arbitrate for the global interconnect similarly to the L1 arbitration. I will assume a very simple bus based global interconnect connected to memory. Since the cluster cache is ported from the L1D design, I can explore similar design tradeoffs using the same parameterized RTL knobs. I chose to keep the cache size constant at 32 kB, but in the evaluation section I will study the effects of having a blocking and a nonblocking cluster cache design.

Figure 5.1 illustrates the cluster cache design. The in/out FIFOs and buses are leading in/out of the cluster cache with a cache ready signal going to the input FIFO, which will provide the cache with a new request if one exists. The input FIFO will notify the bus controller if it can accept further requests. I assumed that the output FIFO will drain fast enough not to miss any responses from the cache. In the future, stall logic should be integrated to prevent the response FIFO from overflowing.

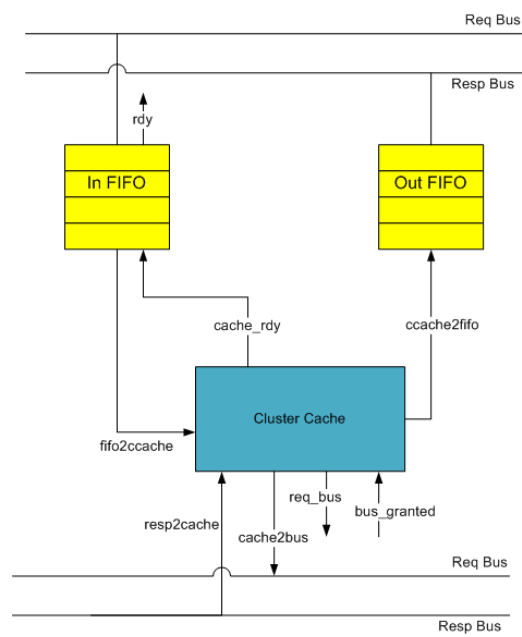


Figure 5.1: Cluster cache diagram

# CHAPTER 6

## EXPERIMENTAL SETUP

Most of my work to evaluate the design exploration was focused on RTL implementation. The main motivation for this is the difficulty in measuring the power/area effects of a design decision using standard architectural modeling commonly employed by architects. It is a daunting task to capture the impact of an architectural feature on the overall design’s power and area with a timing simulator. One can create complex models to try and estimate power and area tradeoffs [23],[24]. This error prone approach is acceptable for some; however, for an aggressive design target such as Rigel, power/area impacts need to be studied in a much more precise fashion when making an architectural decision. As a result I have decided to dedicate the time and effort necessary to create these complex and potentially inaccurate models for the Rigel architecture to implement the RTL and leverage a CAD physical toolflow to collect power, area and performance estimates. Govindan et al. [25] demonstrate the inaccuracies in architectural level power modeling with an end-to-end comparison of power analysis and show that RTL synthesis provides much better results. In addition to the difficulty of modeling power/area tradeoffs in a traditional timing simulator, it is also easy to “cheat” and perform operations that cannot be performed in a single cycle with unrealistic hardware, which results in skewed performance numbers. These mistakes are minimized when collecting performance estimates with an RTL implementation.

### 6.1 Synopsis Toolflow

The CAD flow I have used to get performance, power and area results from my SystemVerilog RTL was the Synopsis toolflow. I have used VCS MX to simulate the RTL, run assembly tests and compiled C benchmarks, in order to

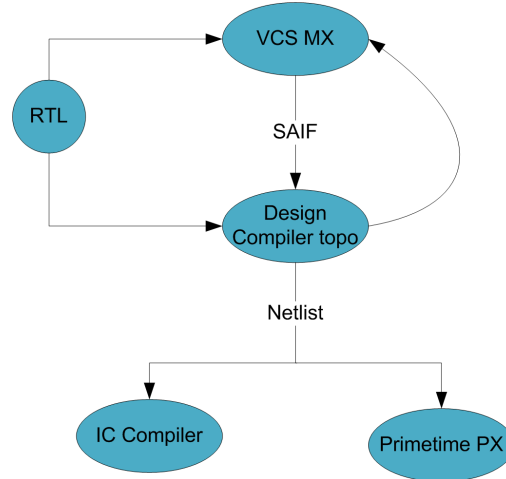


Figure 6.1: Synopsys flow

verify the design and collect performance results for the design. VCS MX also enabled me to collect RTL level switching activity. The simulation collects switching activity factors in a switching activity information file (SAIF) for every port in the design, which later in the process enables me to collect power estimates for a given benchmark. After simulating, I proceed to use Design Compiler-topographical to synthesize my design and create a netlist. I target a production-quality 40 nm high-performance standard cell. Design Compiler is able to provide power and area estimates for a given design. The topographical technology of Design Compiler allows a designer to collect much more accurate area estimates than traditional synthesis technology because it uses coarse placement and routing to guide the area and power calculation. When performing power analysis I provide Design Compiler with the switching activity I collected from the simulation. Design Compiler is able to use the SAIF file and the generated netlist to estimate the average power consumed for a given benchmark. Figure 6.1 illustrates the Synopsys flow.

## 6.2 Simulation Flow

To verify the RTL design and collect performance data, I used SystemVerilog to combine the design under test (DUT) with a functional memory model which simulated a 200 cycle delay.

Figure 6.2(a) shows the interaction between the RTL design and the functional memory model. Notice that they are both written in SystemVerilog; however, the memory model is not synthesizable. This testbench setup allows us to load the Rigel binaries into the memory and have the RTL design run real Rigel compiled code. However, RTL simulation is much slower than C simulation, which makes initialization code problematic. Initialization creates noise in the performance data, but long simulations make that noise negligible. When running shorter simulation in RTL, the init code can significantly impede performance results; as a result we want to omit those effects. To solve this problem I have used the C simulator to execute the initialization code and deliver the results to the RTL in the form of a memory image which we can slurp into the memory model, using the same mechanism we use with the Rigel binary. Figure 6.2(b) shows the simulation flow that enables us not to run the init code in RTL. In practice we achieve different code flows between RTL and C simulator by having the `RigelIsSim()` macro, which checks if a certain special register in the special register file is set. We set the register in the C simulator, but reset it in the RTL code.

Removing the init code in RTL simulation allowed RTL simulation with the following goals:

- Reduce simulation time by orders of magnitude.
- Achieve steady state quickly.
- Enable us to run more complex data-dependent benchmarks such as sparse matrix vector multiply.
- Remove noise in performance results.

## 6.3 Design Exploration Flow

To evaluate a highly variable design with correlated design components, we need an automated and dynamic design exploration flow. Studying a single feature individually may produce not a global optimal design, but a local optimal design. As explained in the introduction section, the core/cache/interconnect decisions are not disjoint design points. If one is trying to find the optimal

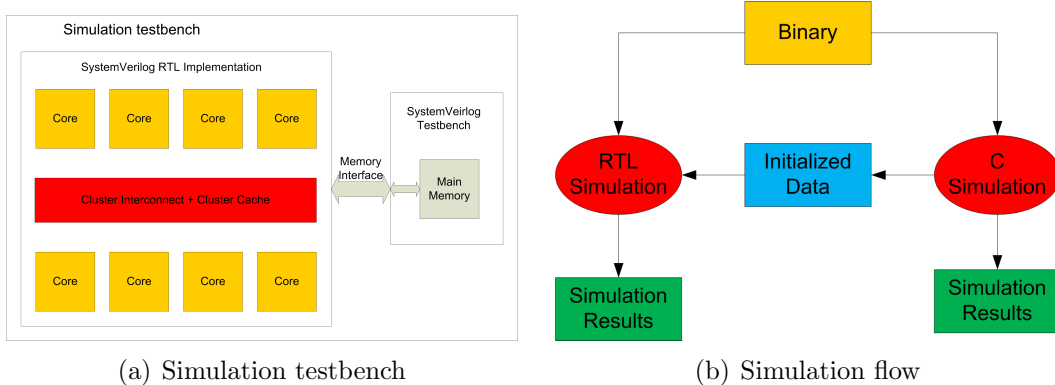


Figure 6.2: Simulation environment

design, he needs to consider their correlation. I will show that when investigating a single feature it can look attractive, but when exploring the entire design space it works out that having that feature produces a suboptimal design choice. On the contrary, a design parameter could seem useless by itself, but when combined with other design choices can be a good design point. For example, a more complex core can exploit the nonblocking L1D producing a performance boost, whereas a simple core can see no benefit from having a nonblocking L1D. The exploration flow is extensible, and allows design variability to easily be added on top of the current design knobs. The flow allows us to control which knobs and values will be examined in the exploration, which allows us to focus on points of interest and speed up results generation. This makes the evaluation of a new design feature and all correlated components fully automated.

To enable a wide RTL design exploration I have created an automated design exploration flow which utilized layers of scripts on top of the Synopsys toolflow. To explore many design points in parallel I have used the Condor distributed computation system on a set of 15 machines. To study the performance of all the design points I needed to simulate all of them. The goal was not to duplicate the RTL for each design, but to reuse the base core RTL. By following a strict SystemVerilog parameterized RTL style [26] the final result was a highly dynamic and configurable RTL I could reuse for all design configurations. The only difference between each design point was a top level configuration file, consisting of a set of macros and parameters. By changing the top level configuration file I was able to reuse, yet completely customize, the RTL design. Employing this technique I simulated a

cross product of all cluster/core configurations. I ran regression tests first to validate the design functionality; upon success, I ran a set of benchmark kernels to collect performance and switching activity data. To study the power and area of each design point I applied the Synopsys synthesis flow on each design. Similarly to the simulation flow, I reused the RTL base, but used a custom configuration file. For every design point I have collected a single area estimation and a power report for each benchmark. Synthesis can add another set of configurable parameters such as clock target, clock gating, technology library, etc.; however, for this work, I kept the synthesis parameters constant. In the future, the flow can be expanded for place and route exploration flow. In the same manner by which the synthesis expands on the simulation and adds another dimension of configuration; place and route exploration flow can add parameters to the synthesis results. Some parameters that could be studied in a place and route flow are floorplans and placement constraints.

## 6.4 RTL Style

Besides the flow described above, proper RTL style is a key component to support the design exploration flow. When developing the RTL I quickly realized that without proper style and thought out interfaces, it will be impossible to cleanly support the automated design exploration I was aiming for. A designer needs to first clearly define module interfaces, so that different versions of the same module can be easily and seamlessly swapped. For features that are within a module or cross module boundaries we need to consider how we can make them dynamic and toggled with a simple flag. As a rule of thumb, unless the feature is clearly necessary, the designer should add the new feature as a parameterized addition rather than a fixed component. The final RTL implementation is highly dynamic and heavily parametrized. Although parametrization is good, we need to be careful not to make the design cluttered and unreadable. There are several techniques that enable design parametrization. The designer needs to choose the appropriate one to make design not only dynamic, but also clean and readable. I will elaborate on the RTL style techniques I have employed.



### 6.4.1 Intra-module macros

The first technique is parametrizing the module name, allowing easy swap between different designs. All swappable modules should have the same interface, sometimes requiring insertion of dummy ports, for ports that are not needed for all modules. Even though that seems wasteful, it grants us the ability to swap modules by simply redefining a macro. For example, I can interchange my L1D design by simply redefining the *L1DCache* macro to either *L1D\_Blocking* or *L1D\_Nonblocking*. Only the nonblocking design has the *stall\_resp* port as discussed in Section 3.2.4; however, in order to provide a clean parameterized design, we have to add it to the blocking design as well. If the port is not used it will be synthesized away by Design Compiler. This style should be utilized when the two modules have significant differences.

### 6.4.2 Inter-module macros

Another RTL style that enables design space flow exploration is the inter-module parametrization. This style should be employed when a feature can be toggled by manipulating only certain sections inside the module. When the inter-module configuration of RTL starts to get cluttered, a good technique is to create a new submodule which will use the intra-module technique above. An example of the inter-module parameterization is the interconnect configuration in the L1D. Depending on the interconnect, the Direct mapped cache and the frontend state machine will be selected.

### 6.4.3 Mixed approach

Some more complex features can span several modules and require a combination of inter-module and intra-module parametrization to enable design feature toggle. For example, to swap from a nonblocking to blocking L1D, I change the L1D module by defining the *L1DCache* macro as *L1D\_blocking*, and define the *L1D\_BLOCKING\_CACHE* macro to propagate the change in the core stall logic, which differs between the two L1D cache designs.

## 6.5 SRAM Hard Macros

All the caches use SRAM cells for storage since they provide a more power and area efficient storage than latches. To evaluate SRAM array area and power I have used SRAMs generated by an ARM memory compiler that targets 45 nm technology. The ARM memory compiler produces a Verilog file for simulation. Simulating the SRAMs with the entire design produces switching activity, which allows more accurate power results than static memory power estimation tools such as CACTI [27]. To synthesize the SRAM hard macros we use the Synopsys database files also produced by the ARM memory compiler and exclude the simulation Verilog files. To support design space exploration I generated pairs of tag and data SRAM arrays for a wide variety of cache sizes. For instruction cache I generated 256kB-4kB arrays with single port SRAMs; for data cache I generated 256kB-32kB arrays with 1 read/1 write port SRAMs. The range is wider for the data cache because I used it for both the L1 and cluster cache. For each data array size I generated the appropriate tag array based on the number of lines in the data array.

# CHAPTER 7

## EVALUATION

In this section I will evaluate a set of cluster design tradeoffs, introduce my benchmarks and present the performance density results. I will finish with a discussion about the effect of each design knob.

### 7.1 Design Power and Area Evaluation

The performance, area and power results will be normalized to the base design results. The base design is the most simplistic design configuration that consists of the smallest L1 caches, no ILP optimizations, blocking cache designs and a bus interconnect. All the design configurations will have a 32 kB cluster cache and 200 cycles memory delay. I synthesized all designs with 500 Mhz clock target, targeting the operating conditions of 0.9 nominal voltage and a temperature of 25 °C. Based on the synthesis results, the base cluster design area is 1.2 mm<sup>2</sup> with an average power consumption of 140 mW. Figure 7.1(a) shows the major contributors of the core components to the total core area and power of the core. Figure 7.1(b) shows the major contributions of the cluster components to the total cluster power and area.

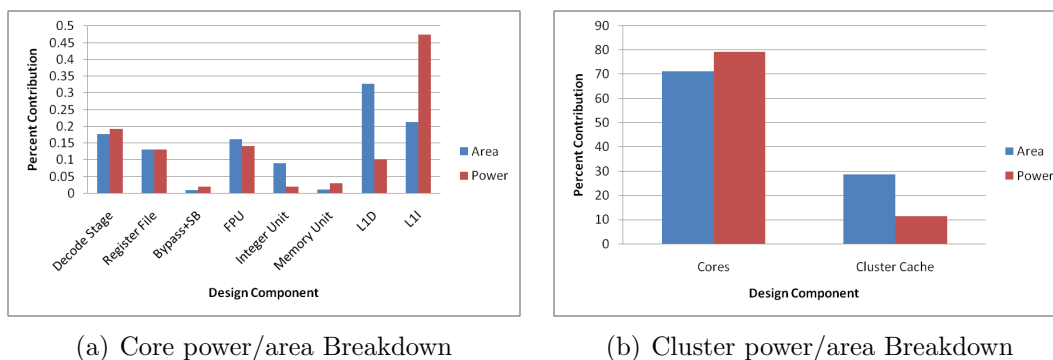


Figure 7.1: Relative power/area contribution

## 7.2 Cache Verification

To verify the cache I used a hands on approach. I chose to verify the cache functionality with Rigel code execution. That allowed me to efficiently verify both cache functionality and its integration with the core, while developing real Rigel code. I used both small directed tests and larger benchmark codes to verify the cache and its integration with the core. To verify execution I compared register file changes between simulator and RTL execution. Also, for both core and cluster RTL simulation I verified the final register file of the RTL execution with the simulator output.

## 7.3 Design Space Exploration

In my evaluation I studied the effect of several design choices:

**L1 Data/Instruction Cache Size** adds fairly predictable area and power; however, it is not clear if the cost is recovered in performance.

**L1 Data Cache Design** Varying the cache design between a blocking and a nonblocking design. This knob has slight power/area effect, and with core MLP has potential to be a good contributor.

**ILP Optimization** Optimization helps the core to expose more ILP, which also means more memory operations are issued increasing MLP in the system. More specifically we enable a full bypass network removing any false dependencies. Also, we use the full bypass as a scoreboard, allowing us to remove output dependencies.

**Cluster Cache Design** is similar to the L1D cache design, in terms of cost and potential; however, the MLP comes from memory streams from eight cores in the cluster, making the MLP more abundant and completely independent which suits well the current operation of the non-blocking cache, as described in Section 3.2.4. The area and power impact of having a different cluster cache design does affect the system differently based on the cluster interconnect design choice. A bus will have a centralized cache, so only one instance of the structures that are required for nonblocking cache support are required. However, in case

of a crossbar there are essentially eight independently operating banks, which requires replication of the control structures for each bank.

**Cluster Interconnect** The interconnect architecture is a simple bus or a complex crossbar.

## 7.4 Evaluation Metrics

As part of my work I realized that the standard performance evaluations are not sufficient and cannot show us the true impact of an architectural decision on a power/area constrained system such as Rigel. I decided to present my results in terms of performance per area unit and performance per power unit. Also, for clear analysis it is necessary to present the performance, power and area contribution of a design point in order to understand why a certain design has a low performance density: Is it due to high cost or low return in performance? For each design knob, I presented the performance, power area when compared to a less promising design point. The less promising design choices are the characteristics of the base design described above, replacing the bus with the crossbar. Each design knob effect is represented by a set of points, where each point is constructed as described above, while keeping all other design choices constant.

## 7.5 Benchmarks

I have written several benchmarks to run on my RTL implementation and the Rigel simulator. I have used a static work partitioning programming using the `RigelGetCoreNum()` macro. Each core will execute a different iteration of the inner loop based on the core number. Using the simulation flow and the `RigelIsSim()` macro described in the environment section, I reach steady state execution extremely fast.

### 7.5.1 Dense matrix multiplication

Dense matrix multiplication (DMM) is a common operation in scientific applications and a common benchmark for parallel systems. The work done in

DMM is illustrated in Figure 7.2. DMM is embarrassingly parallel, which means it is straightforward to parallelize the computation. Each core is assigned a square tile that is equal to the total work divided by eight. The simulation ran on matrices of size 32x32. Figure 7.3 shows the results for the DMM benchmark.

### 7.5.2 FIR filter

Finite impulse response (FIR) is a common algorithm in the embedded world. The algorithm performs a sequence of dot product operations. Each core will get an eighth of the number of inputs. Figure 7.4 shows the results for the FIR benchmark.

### 7.5.3 Sobel edge detection

Sobel edge detection is a common algorithm used in computer vision for feature detection and feature extraction. The algorithm uses convolution to detect sharp changes in color. Figure 7.5 illustrates the result of running Sobel on an image. Sobel edge detection is also embarrassingly parallel. The work is statically divided between all the cores. Each core will execute the total amount of rows divided by the number of cores. In my simulation I have used an image of size 64x64 pixels. Figure 7.6 shows the results for the Sobel benchmark.

### 7.5.4 Large stride

This kernel is a completely synthetic benchmark I created to illustrate traversal of large structures, which produces very bad locality, rendering the L1 useless. The benchmark is meant to stress the limits of the interconnect and cache design tradeoffs. Figure 7.7 shows the results for the large stride traversal benchmark.

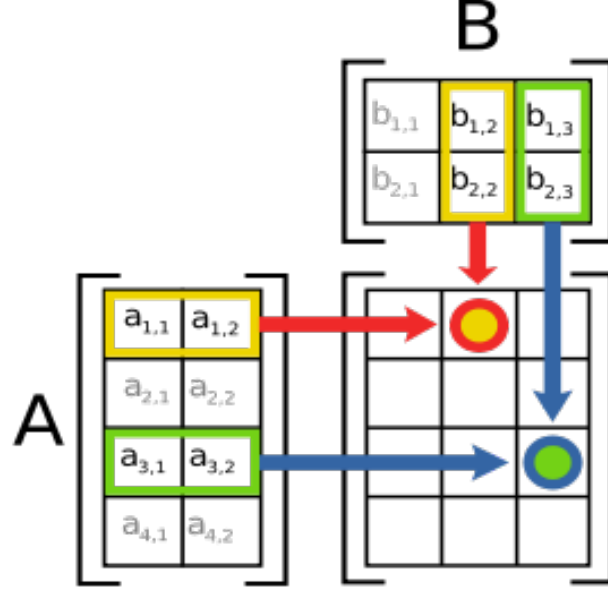
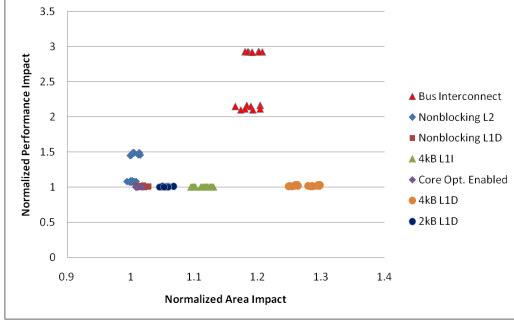


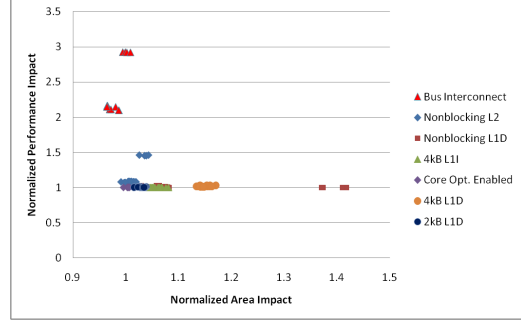
Figure 7.2: DMM illustration

### 7.5.5 Sparse matrix dense vector multiplication

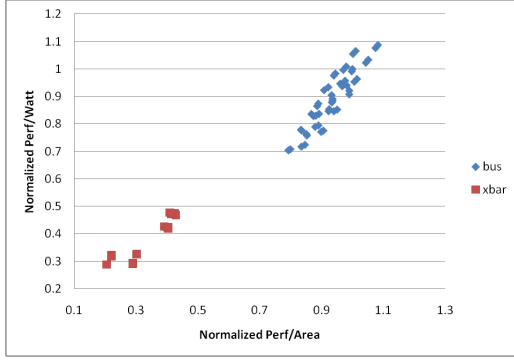
Sparse matrices are mostly empty, but contains nonzero elements throughout, as can be seen in Figure 7.8. Employing dense algorithms on sparse matrices can be very inefficient and sometimes impractical for large sparse matrices. To store these matrices we use a dynamic structure which contains a list of elements with a list of indices pointing to the element list. This specific benchmark used the Yale sparse matrix format. The nature of the data structure that holds the sparse matrix creates an irregular data access which reduces the cache hit rate. Also, dividing the work statically creates load balancing issues, since the amount of work for every tile of the matrix can have different numbers of nonzero elements. This benchmark performs sparse matrix dense vector multiplication (SPMVM) as illustrated in Figure 7.8. Although simple, this benchmark captures the tradeoffs of sparse matrix applications. This benchmark is the longest and most intensive of all the above, and it captures a wide spectrum of the design tradeoffs. Figure 7.9 shows the results for the SPMVM benchmark.



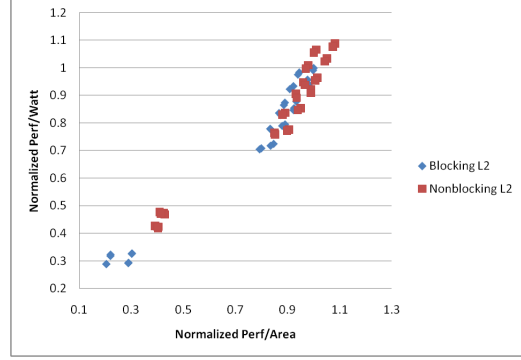
(a) Normalized Performance/Area Impact



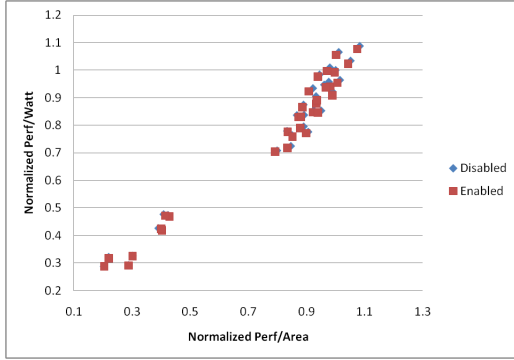
(b) Normalized Performance/Power Impact



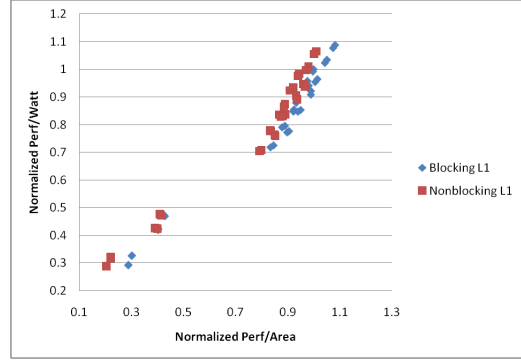
(c) Normalized Interconnect Impact



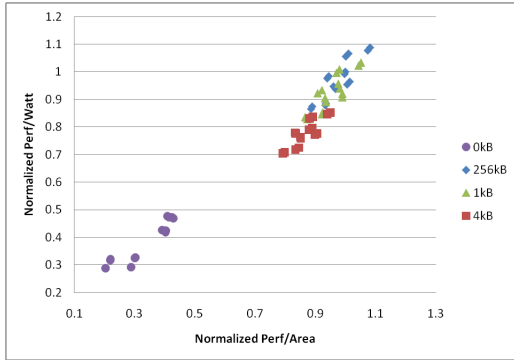
(d) Normalized L2 Design Impact



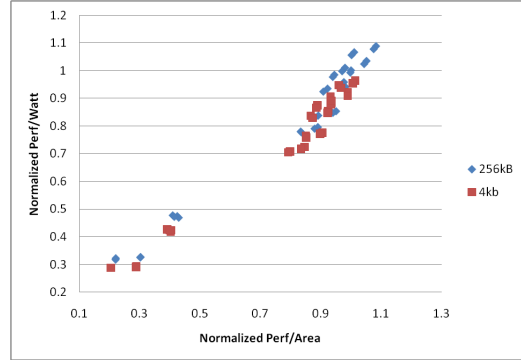
(e) Normalized Core Optimization Impact



(f) Normalized L1D Design Impact



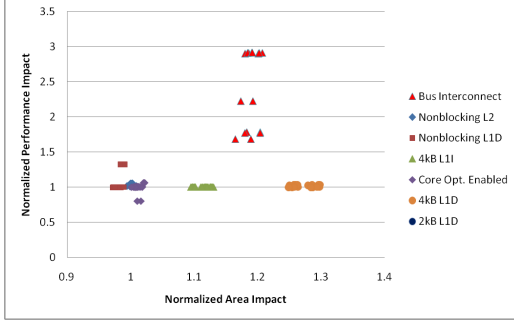
(g) Normalized L1D Size Impact



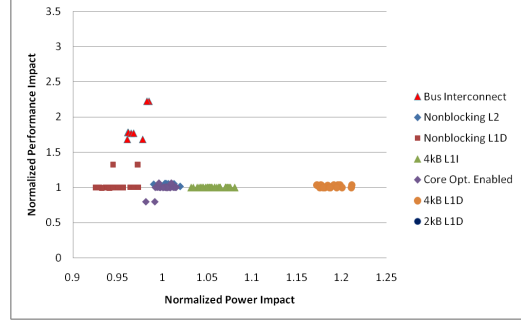
(h) Normalized L1I Size Impact

Figure 7.3: DMM results

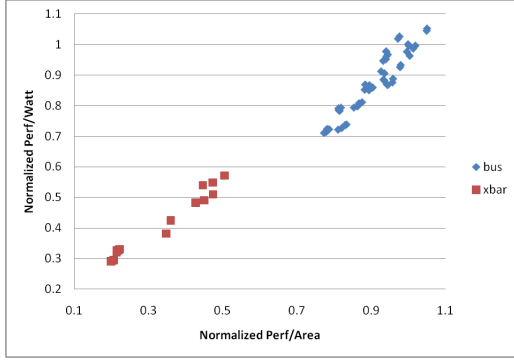




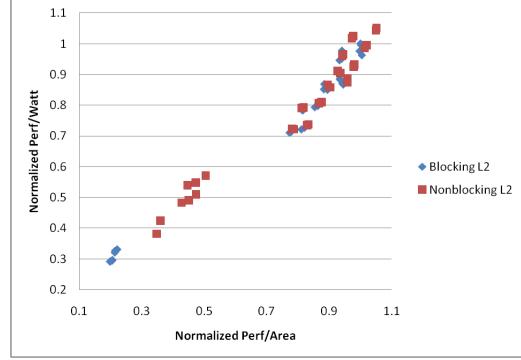
(a) Normalized Performance/Area Impact



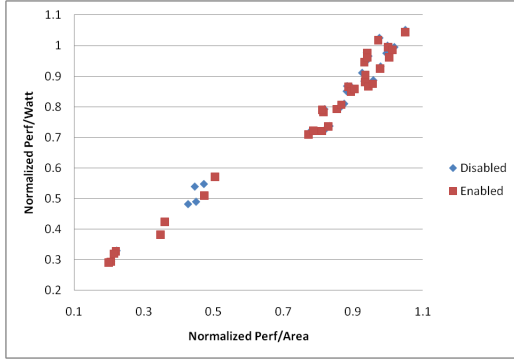
(b) Normalized Performance/Power Impact



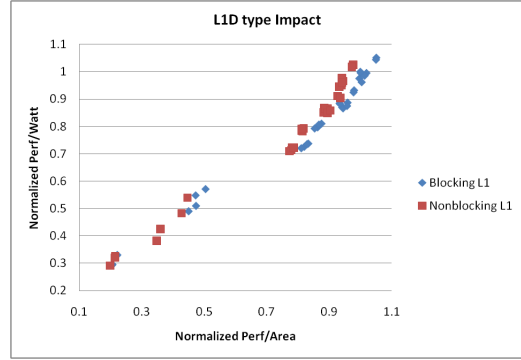
(c) Normalized Interconnect Impact



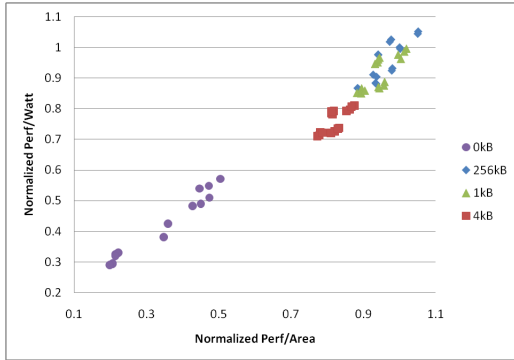
(d) Normalized L2 Design Impact



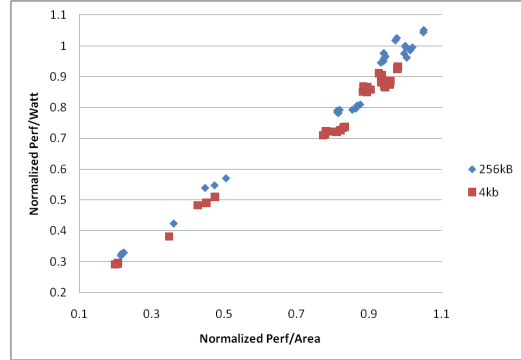
(e) Normalized Core Optimization Impact



(f) Normalized L1D Design Impact



(g) Normalized L1D Size Impact



(h) Normalized L1I Size Impact

Figure 7.4: FIR results

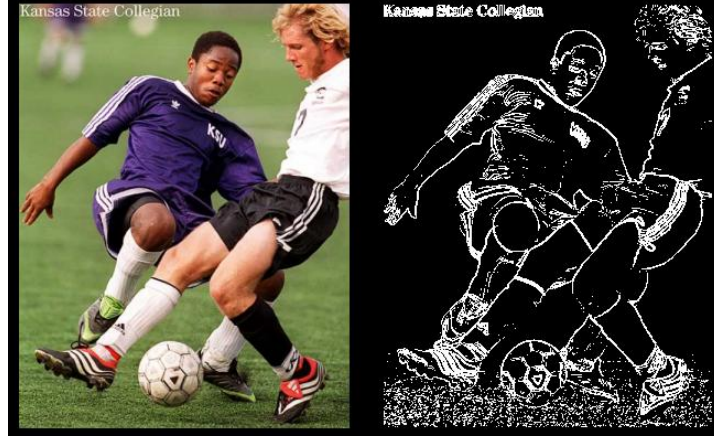


Figure 7.5: Sobel edge detection illustration

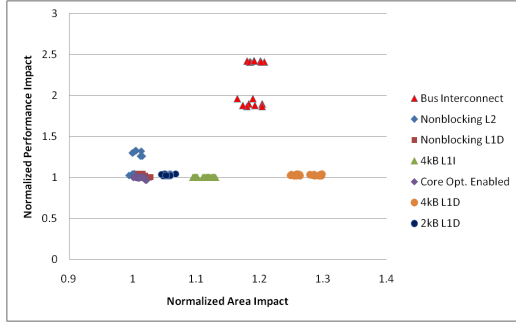
## 7.6 Results

From the results we can make the following observations for each design knob:

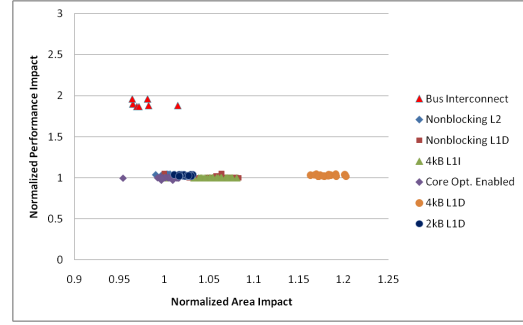
**L1 Instruction Cache Size** Having a larger L1I does not provide any performance benefit. All the benchmarks have a small main inner loop which can be completely contained in a small L1. Having a bigger L1I just results in a power/area penalty which makes 256B L1I the most attractive design point.

**L1 Data Cache Size** All the benchmarks except SPMVM, the most complex benchmark, show no performance benefit for having a larger L1D. This is due to the fact that the benchmarks have a very regular data access. As a result, for most of the benchmarks, having a 256B L1D is the best design choice. However, SPMVM, dealing with sparse matrix structures, observes a performance gain due to a larger cache. For SPMVM the best design choice is the 1 kB cache, since it provides the most dense performance per unit of area and power. Although the 4 kB cache has better performance than the 1 kB cache, it incurs much higher area/power costs.

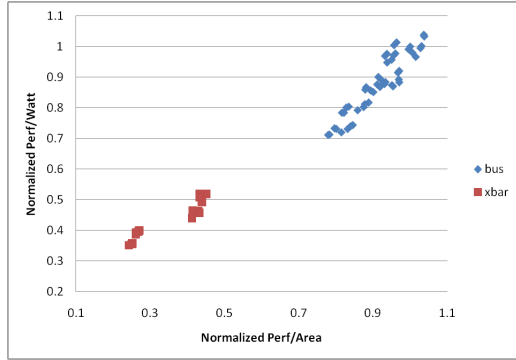
We can see an interesting effect in 7.9(g), where larger cache size results in lower energy consumption. This effect is the result of lower frequency in cluster cache access, which results in higher interconnect and cluster cache power.



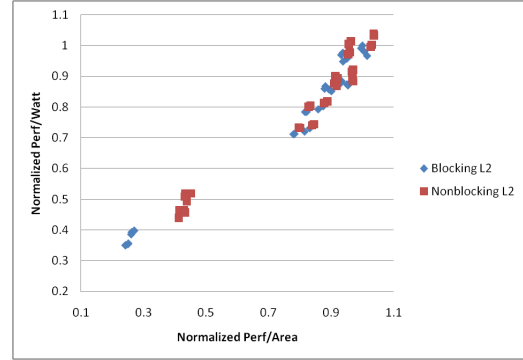
(a) Normalized Performance/Area Impact



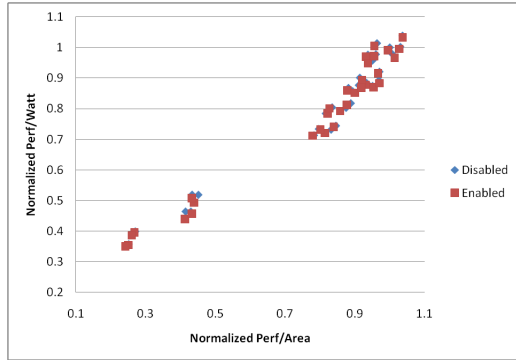
(b) Normalized Performance/Power Impact



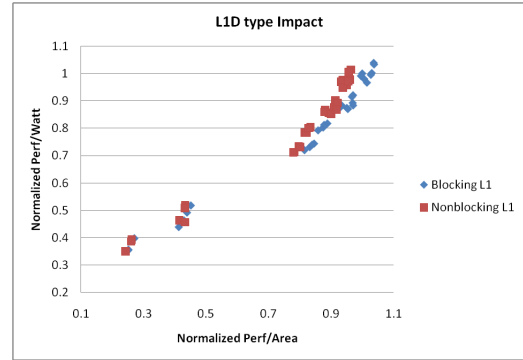
(c) Normalized Interconnect Impact



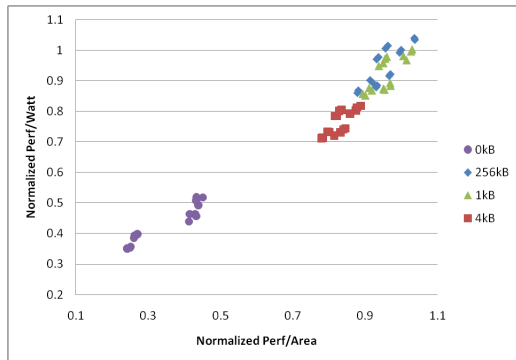
(d) Normalized L2 Design Impact



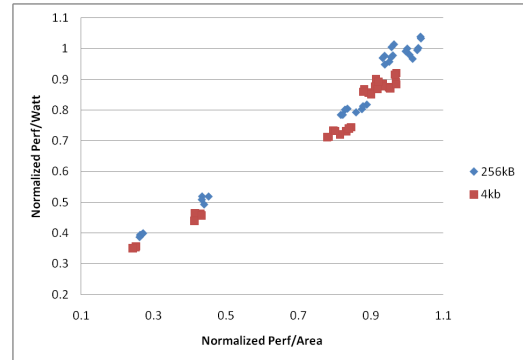
(e) Normalized Core Optimization Impact



(f) Normalized L1D Design Impact

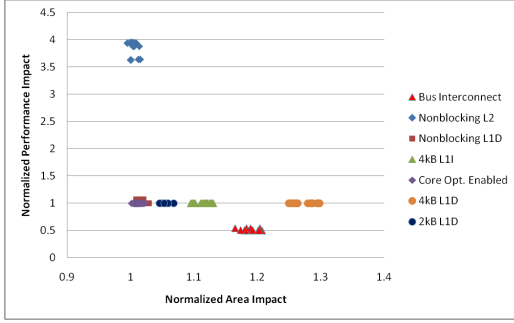


(g) Normalized L1D Size Impact

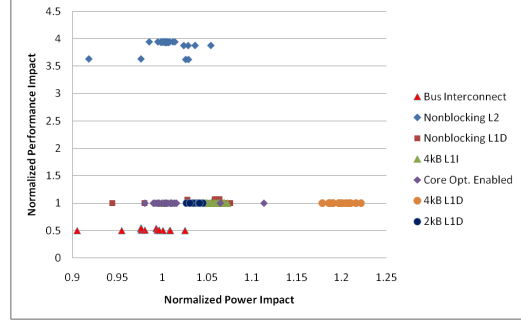


(h) Normalized L1I Size Impact

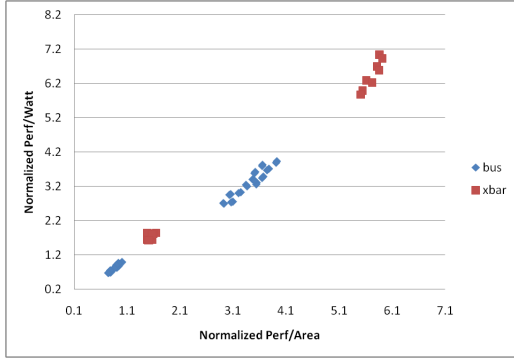
Figure 7.6: Sobel results



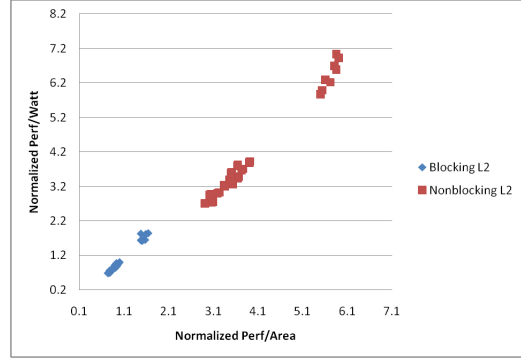
(a) Normalized Performance/Area Impact



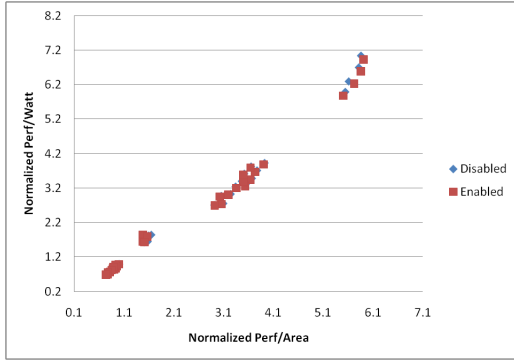
(b) Normalized Performance/Power Impact



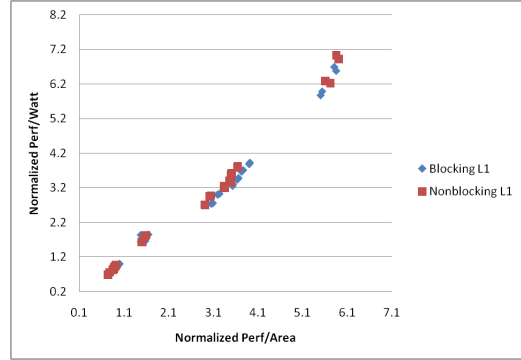
(c) Normalized Interconnect Impact



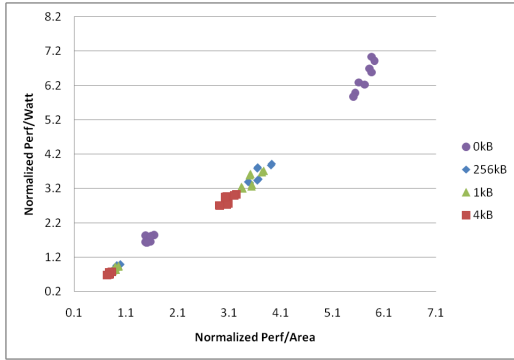
(d) Normalized L2 Design Impact



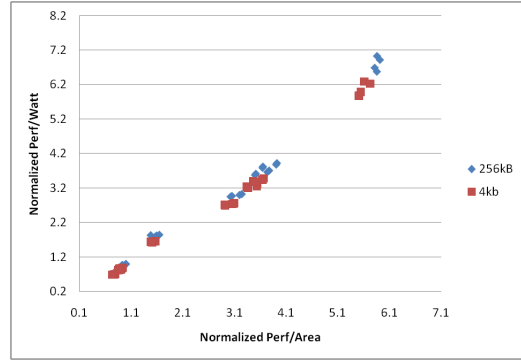
(e) Normalized Core Optimization Impact



(f) Normalized L1D Design Impact



(g) Normalized L1D Size Impact



(h) Normalized L1I Size Impact

Figure 7.7: Large stride results

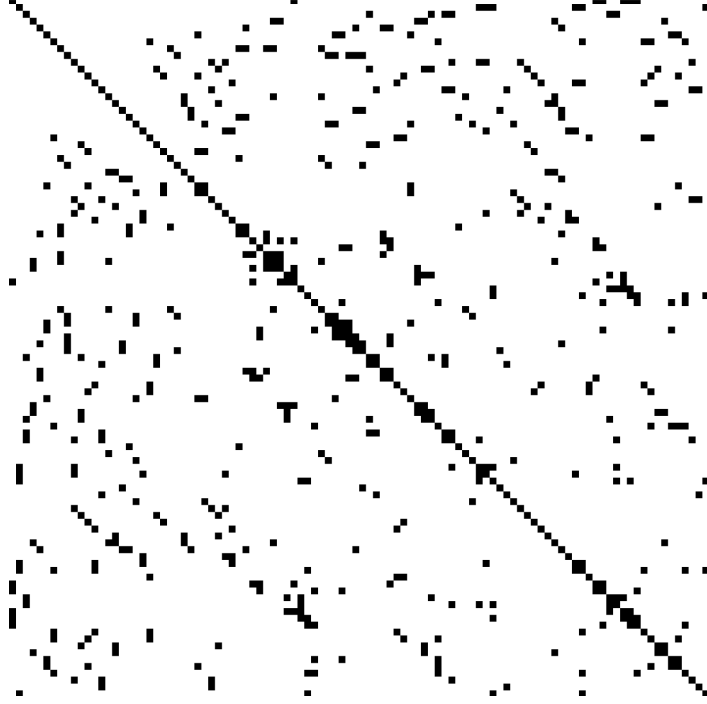
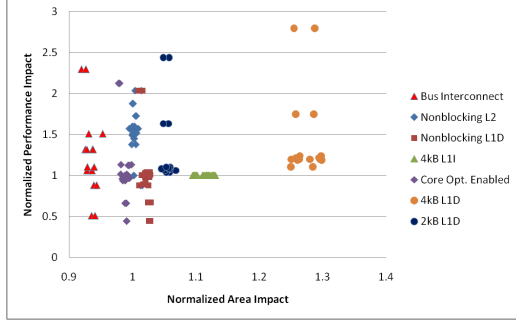


Figure 7.8: SPMVM illustration

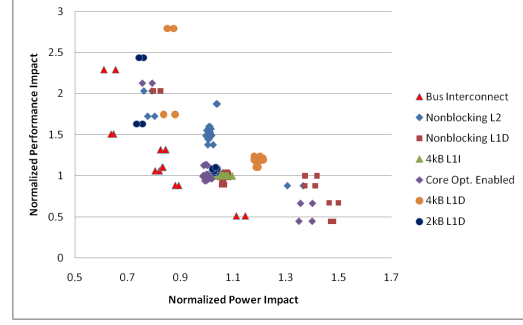
**L1 Data Cache Design** All the benchmarks except SPMVM show no performance benefit for having a complex nonblocking cache. There are two main reasons for this result. First, the in-order execution limits the MLP and does not allow full utilization of the nonblocking feature. Second, the design decision of blocking the request if another request to the same line is pending creates a stall for all other independent memory instructions. Further compiler and hardware optimizations can help alleviate this issue.

Although SPMVM did show a relative performance gain in Figure 7.9(b) with a nonblocking cache, the best design point for SPMVM does not include a nonblocking L1D as seen in Figure 7.9(f). All benchmarks agree that, currently, the best L1D design is a blocking cache.

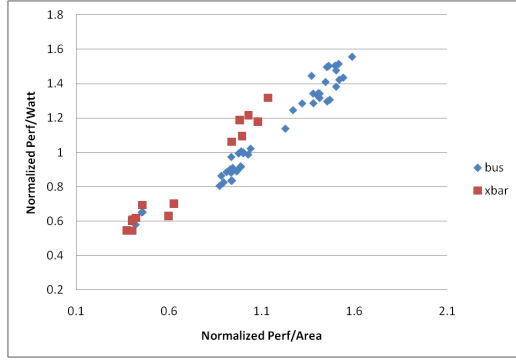
**ILP Optimization** Similarly to the cache design, only the SPMVM benchmark shows a potential performance benefit from having ILP optimizations. The main reason for that is the fact that the compiler does a good job removing false and output dependencies. SPMVM has a much longer and more complex inner loop, which can benefit from the ILP optimizations. The negligible effect on power and area and the potential



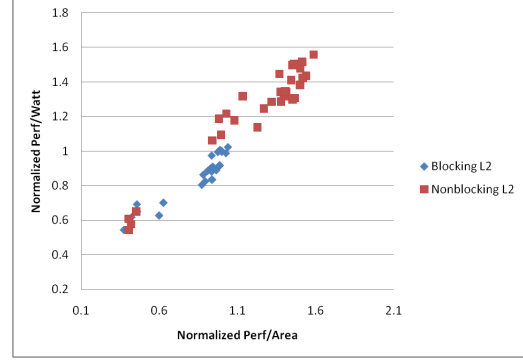
(a) Normalized Performance/Area Impact



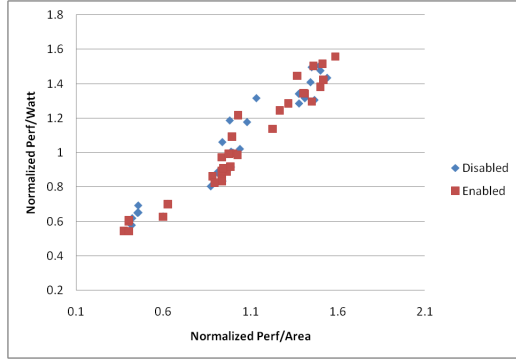
(b) Normalized Performance/Power Impact



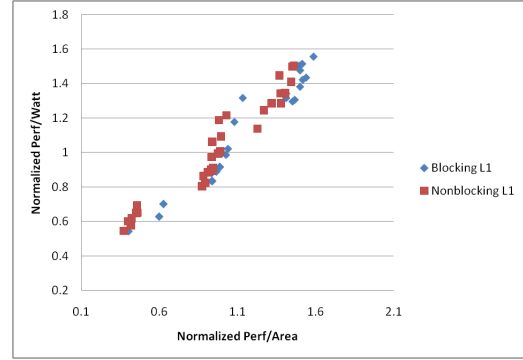
(c) Normalized Interconnect Impact



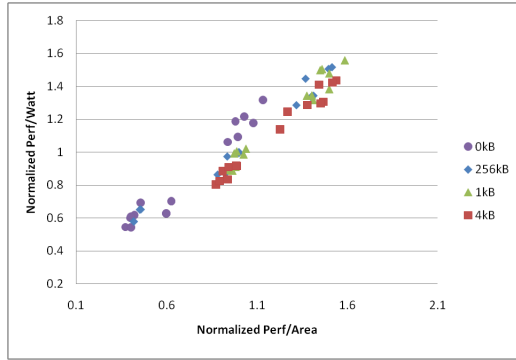
(d) Normalized L2 Design Impact



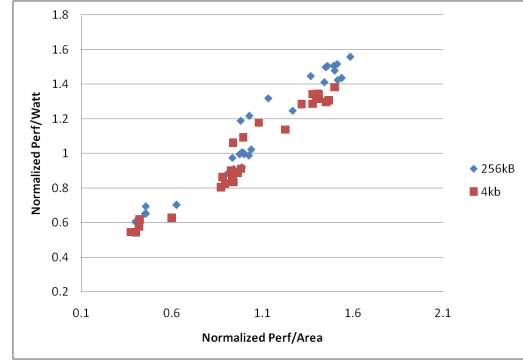
(e) Normalized Core Optimization Impact



(f) Normalized L1D Design Impact



(g) Normalized L1D Size Impact



(h) Normalized L1I Size Impact

Figure 7.9: SPMVM results

performance benefit that can be seen in SPMVM make it a desirable design decision.

**Cluster Cache Design** All benchmarks observed performance gain from a nonblocking L2, while having a minimal power/area impact. The performance gain varied greatly, but since we essentially do not pay area and power cost, the nonblocking potential benefit makes a viable design decision. The nonblocking cache performed better due to the abundance of independent memory streams from the eight cores in the cluster and the high memory latency penalty. It is crucial to send the request to memory as soon as possible, and the nonblocking cache parallelizes cache misses. FIR sees almost no performance benefit due to high locality of reference, while the large stride kernel sees 4X performance gain due to the high cluster cache miss rate.

**Cluster Interconnect** All the benchmarks except large stride prefer the bus interconnect with a privatized L1. The main motivation for having a crossbar was removing contention while reducing area and power. However, I observe that interconnect contention issues diminish by the global memory latencies. In fact, higher latency, simplicity of the core and the suboptimal code result in poor crossbar performance. In terms of power and area tradeoffs, the crossbar also does not deliver. The crossbar complicates routing, and duplication in cluster cache logic results in high power/area cost and no benefit over having an L1D. The later problem will be exacerbated with a more complex cluster cache. It is worth mentioning that I believe further code optimizations can bring the crossbar performance closer to the bus interconnect.

The only benchmark that prefers the crossbar interconnect is the large stride benchmark since it synthetically creates high contention and a high L1 miss. Considering that, the performance improvement is only 2X. I choose to focus on the more realistic benchmark results. All the above make the bus interconnect the easier, more straightforward and attractive design point.

# CHAPTER 8

## CONCLUSION

From my evaluation I observe that relatively simple design choices are best for a throughput oriented design such as Rigel. My results show that the following design decisions produce the performance density:

- Bus interconnect
- Nonblocking cluster cache
- Blocking L1D
- Smallest L1 caches
- ILP optimizations enabled

I evaluated the design space for the memory hierarchy in a Rigel cluster. Memory system design space is large, and I am far from studying all design points, nor was that my intent. With the presented methodology and RTL style one can effectively explore the performance, power and area effects of a certain design point of the system. This methodology should be used throughout the entire design, allowing a hardware designer to reinspect the effect of changing a single component on all correlated design decisions. One might discover that previous design decisions no longer apply. An example of that can be my observation that having a nonblocking L1D gives us no performance benefits since we cannot extract enough parallelism from our simple in order core. If the Rigel project decides to go back and reevaluate the core design and make it more complex—i.e., out-of-order execution or a multithreaded core—the nonblocking L1D can have a much more significant impact.

This is drastically different from the approach of picking a design point for each component of your system based its individual performance, because it considers the correlation between design components and real perfor-



mance/power and performance/area impact. This approach is RTL centric, which results in more time to evaluate a design feature, however, the results are much more accurate. Also, it requires careful RTL coding and a much larger RTL code base. This methodology can benefit the commercial chip industry. It can enable a chip company to be more dynamic and allow them to pick the right chip design based on the costumers target application.

## REFERENCES

- [1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, March 2007.
- [2] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.
- [3] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, and B. Baas, “Asap: An asynchronous array of simple processors,” *IEEE Journal of Solid-State Circuits*, vol. 43, no. 3, pp. 695–705, 2008.
- [4] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukolun, “The stanford hydra cmp,” *IEEE Micro*, vol. 20, no. 2, pp. 71–84, 2000.
- [5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, pp. 40–53, March 2008. [Online]. Available: <http://doi.acm.org/10.1145/1365490.1365500>
- [6] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, “Rigel: An architecture and scalable programming interface for a 1000-core accelerator,” in *Proceedings of the International Symposium on Computer Architecture*, June 2009, pp. 140–151.
- [7] R. Kumar, V. Zyuban, and D. Tullsen, “Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling,” in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005, pp. 408–419.
- [8] N. E. Jerger and L.-S. Peh, “On-chip networks,” *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–141, 2009. [Online]. Available: <http://www.morganclaypool.com/>
- [9] P. Stenstrom, “A survey of cache coherence schemes for multiprocessors,” *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

- [10] S. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.
- [11] T. D. Givargis, J. Henkel, and F. Vahid, “Interface and cache power exploration for core-based embedded system design,” in *Proceedings of the 1999 IEEE/ACM International Conference on Computer-Aided Design*, 1999, pp. 270–273.
- [12] C.-L. Su and A. M. Despain, “Cache design trade-offs for power and performance optimization: A case study,” in *Proceedings of the 1995 International Symposium on Low Power Design*, 1995, pp. 63–68.
- [13] A. Ghosh and T. Givargis, “Analytical design space exploration of caches for embedded systems,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 650–655.
- [14] N. Jouppi and S. Wilton, “Tradeoffs in two-level on-chip caching,” in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994, pp. 34–45.
- [15] S. Przybylski, M. Horowitz, and J. Hennessy, “Performance tradeoffs in cache design,” in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 290–298.
- [16] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel, “Tradeoffs in designing accelerator architectures for visual computing,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 164–175.
- [17] O. Azizi, A. Mahesri, S. J. Patel, and M. Horowitz, “Area-efficiency in *CMP* core design: Co-optimization of microarchitecture and physical design,” *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 56–65, July 2009. [Online]. Available: <http://doi.acm.org/10.1145/1577129.1577138>
- [18] M. Monchiero, R. Canal, and A. Gonzalez, “Power/performance/thermal design-space exploration for multi-core architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 5, pp. 666–681, May 2008.
- [19] Tensilica, “Xtensa microprocessor: Overview handbook,” 2002. [Online]. Available: <http://www.tensilica.com/>
- [20] M. Mohiyuddin, M. Murphy, L. Oliker, J. Shalf, J. Wawrzynek, and S. Williams, “A design methodology for domain-optimized power-efficient supercomputing,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–12.

- [21] N. P. Jouppi, “Cache write policies and performance,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993, pp. 191–201.
- [22] J. Clabes, J. Friedrich, M. Sweet, J. Dilullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson, “Design and implementation of the *POWER5* microprocessor,” in *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, February 2004, pp. 56–57.
- [23] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, “The design and use of simplepower: A cycle-accurate energy estimation tool,” in *Proceedings of the 37th Annual Design Automation Conference*, 2000, pp. 340–345.
- [24] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 83–94.
- [25] M. S. S. Govindan, S. W. Keckler, and D. Burger, “End-to-end validation of architectural power models,” in *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, 2009, pp. 383–388.
- [26] C. E. Cummings, “New Verilog-2001 techniques for creating parameterized models (or down with ‘define and death of a defparam!’),” HDL-CON, 2002.
- [27] S. Wilton and N. Jouppi, “Cacti: An enhanced cache access and cycle time model,” *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.